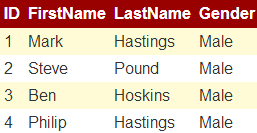


**1.** LINQ query can be written using any .NET supported programming language  
**2.** LINQ provider is a component between the LINQ query and the actual data source, which converts the LINQ query into a format that the underlying data source can understand. For example LINQ to SQL provider converts a LINQ query to T-SQL that SQL Server database can understand.

**For example**, the application that we are developing should display male students in a GridView control as shown below.    
   
  
**To achieve this**  
**Step 1:**We first create the required table

Create Table Students

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50)

)

GO

Insert into Students values ('Mark', 'Hastings', 'Male')

Insert into Students values ('Steve', 'Pound', 'Male')

Insert into Students values ('Ben', 'Hoskins', 'Male')

Insert into Students values ('Philip', 'Hastings', 'Male')

Insert into Students values ('Mary', 'Lambeth', 'Female')

GO

**Step 2:** Write the required ADO.NET code to retrieve data from SQL Server database as shown below.

using System;

using System.Collections.Generic;

using System.Configuration;

using System.Data.SqlClient;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        {

            string cs = ConfigurationManager.ConnectionStrings["DBCS"].ConnectionString;

            SqlConnection con = new SqlConnection(cs);

            SqlCommand cmd = new SqlCommand

                ("Select ID, FirstName, LastName, Gender from Students where Gender='Male'", con);

            List<Student> listStudents = new List<Student>();

            con.Open();

            SqlDataReader rdr = cmd.ExecuteReader();

            while (rdr.Read())

            {

                Student student = new Student();

                student.ID = Convert.ToInt32(rdr["ID"]);

                student.FirstName = rdr["FirstName"].ToString();

                student.LastName = rdr["LastName"].ToString();

                student.Gender = rdr["Gender"].ToString();

                listStudents.Add(student);

            }

            con.Close();

            GridView1.DataSource = listStudents;

            GridView1.DataBind();

        }

    }

    public class Student

    {

        public int ID { get; set; }

        public string FirstName { get; set; }

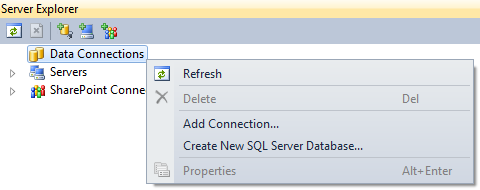
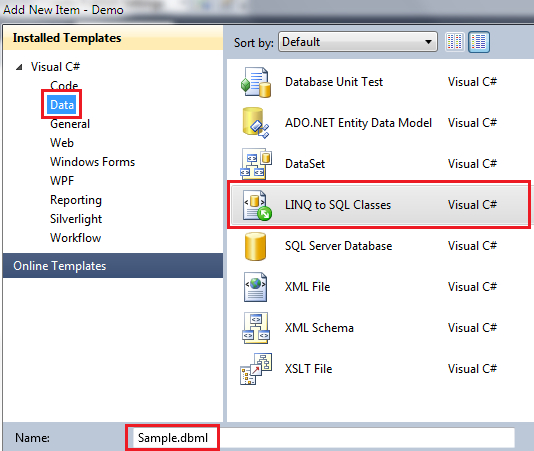
        public string LastName { get; set; }

        public string Gender { get; set; }

    }

}

If we misspell table or column names in the SQL Query, we will not know about it at compile time. At run time the page crashes and that's when we will know about this error. Also notice that there is no **intellisense**when typing table and column names. Misspelled column names when reading from the reader will also cause the same problem. With LINQ we will have intellisense and compile time error checking.  
  
**Now let's achieve the same thing using LINQ to SQL.**

**Step 1:** Create a new empty asp.net web application and name it **Demo**  
  
**Step 2:**Click on **"View"**menu item and select **"Server Explorer"**  
  
**Step 3:**In **"Server Explorer"**window, right click on **"Data Connections"**and select **"Add Connection"**option   
   
  
**Step 4:** Specify your SQL Server name and the credentials to connect to SQL Server. At this point we should be connected to SQL Server from Visual Studio.  
  
**Step 5:** Adding **LINQ to SQL Classes**  
**a)** Right click on the **"Demo"**project in solution explorer and select **"Add New Item"**option  
**b)** In the **"Add New Item"**dialog box, select **"Data"**under **"Installed Templates"**  
**c)** Select **"LINQ to SQL Classes"**  
**d)** Set **Name = Sample.dbml**  
**e)** Finally click **"Add"**button   
   
  
**Step 6:** From **"Server Explorer"**window drag and drop **"Students"**table onto **"Sample.dbml"**designer file.  
  
**Step 7:**Add a webform. Drag and drop a gridview control.  
  
**Step 8:**Copy and paste the following code in the code-behind file

using System;

using System.Linq;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        {

            SampleDataContext dataContext = new SampleDataContext();

            GridView1.DataSource = from student in dataContext.Students

                                   where student.Gender == "Male"

                                   select student;

            GridView1.DataBind();

        }

    }

}

Notice that, with LINQ we are getting intellisense. If we misspell the table or column names we will get to know about them at compile time. Open SQL Profiler. Run the application, and notice the SQL Query that is generated.

To write LINQ queries we use the **LINQ Standard Query Operators**. The following are a few Examples of Standard Query Operators  
select  
from  
where   
orderby   
join  
groupby   
  
   
  
**There are 2 ways to write LINQ queries using these Standard Query Operators**  
**1.** **Using Lambda Expressions.** We discussed Lambda Expressions in detail in [Part 99](http://www.youtube.com/watch?v=LDgQ-spnrYY&list=PLAC325451207E3105&index=100)of [C# Tutorial](http://www.youtube.com/playlist?list=PLAC325451207E3105)  
  
**2.** **Using SQL like query expressions**  
  
The **Standard Query Operators**are implemented as extension methods on IEnumerable<T> interface. We will discuss, what extension methods are and how to implement them in a later video session.  
  
**For now let's focus on the 2 ways of writing a LINQ query**. From a performance perspective there is no difference between the two. Which one to use depends on your personal preference. But keep in mind, behind the scene, LINQ queries written using SQL like query expressions are translated into their lambda expressions before they are compiled.   
  
We will use the following Student class in this demo. GetAllStudents() is a static method that returns List<Student>. Since List<T> implements IEnumerable<T>, the LINQ Standard Query Operators will be available and can be applied on List<Student>.

public class Student

{

    public int ID { get; set; }

    public string Name { get; set; }

    public string Gender { get; set; }

    public static List<Student> GetAllStudents()

    {

        List<Student> listStudents = new List<Student>();

        Student student1 = new Student

        {

            ID = 101,

            Name = "Mark",

            Gender = "Male"

        };

        listStudents.Add(student1);

        Student student2 = new Student

        {

            ID = 102,

            Name = "Mary",

            Gender = "Female"

        };

        listStudents.Add(student2);

        Student student3 = new Student

        {

            ID = 103,

            Name = "John",

            Gender = "Male"

        };

        listStudents.Add(student3);

        Student student4 = new Student

        {

            ID = 104,

            Name = "Steve",

            Gender = "Male"

        };

        listStudents.Add(student4);

        Student student5 = new Student

        {

            ID = 105,

            Name = "Pam",

            Gender = "Female"

        };

        listStudents.Add(student5);

        return listStudents;

    }

}

The **LINQ query**should return just the **Male**students.   
  
**LINQ query using Lambda Expressions.**

IEnumerable<Student>students=Student.GetAllStudents**()**

.Where**(**student=>student.Gender=="Male"**);**  
  
**LINQ query using using SQL like query expressions**

IEnumerable<Student>students=fromstudentinStudent.GetAllStudents**()**

wherestudent.Gender=="Male"

selectstudent**;**  
  
**To bind the results of this LINQ query to a GridView**  
GridView1.DataSource = students;  
GridView1.DataBind();

**What are Extension Methods**  
**According to MSDN**, Extension methods enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.   
**Extension methods are a special kind of static method**, but they are called as if they were instance methods on the extended type.   
  
**For client code written in C# and Visual Basic**, there is no apparent difference between calling an extension method and the methods that are actually defined in a type.  
  
**Let us understand what this definition actually means.**  
**LINQ's standard query**operators (select, where etc ) are implemented in Enumerable class as extension methods on the IEnumerable<T> interface.  
  
**Now look at the following query**

List<int> Numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> EvenNumbers = Numbers.Where(n => n % 2 == 0);

In spite of **Where()**method not belonging to **List<T>**class, we are still able to use it as though it belong to **List<T>**class. This is possible because **Where()**method is implemented as extension method in **IEnumerable<T>**interface and **List<T>**implements **IEnumerable<T>**interface.  
  
**How to implement extension methods**  
We want to define a method in the string class (let's call it ChangeFirstLetterCase), which will change the case of the first letter of the string. For example, if the first letter of the string is lowercase the function should change it to uppercase and viceversa.  
  
We want to be able to call this function on the string object as shown below.

string result = strName.ChangeFirstLetterCase();

Defining **ChangeFirstLetterCase**() method directly in the **string**class is not possible as we don't own the string class. It belongs to .NET framework. Another alternative is to write a wrapper class as shown below.

public class StringHelper

{

    public static string ChangeFirstLetterCase(string inputString)

    {

        if (inputString.Length > 0)

        {

            char[] charArray = inputString.ToCharArray();

            charArray[0] = char.IsUpper(charArray[0]) ?

                char.ToLower(charArray[0]) : char.ToUpper(charArray[0]);

            return new string(charArray);

        }

        return inputString;

    }

}

Wrapper class works, but the problem is, we cannot call **ChangeFirstLetterCase**() method using the following syntax.  
string result = strName.ChangeFirstLetterCase();  
  
Instead we have to call it as shown below.  
string result = StringHelper.ChangeFirstLetterCase(strName);  
  
Convert **ChangeFirstLetterCase**() method to an extension method to be able to call it using the following syntax, as though it belongs to string class.  
string result = strName.ChangeFirstLetterCase();  
  
To **convert ChangeFirstLetterCase() method to an extension method**, make the following 2 changes  
**1.** Make StringHelper static class  
**2.** The type the method extends should be passed as a first parameter with this keyword preceeding it.  
  
With these 2 changes, we should be able to call this extension method in the same way we call an instance method. Notice that the extension method shows up in the intellisense as well, but with a different visual clue.  
string result = strName.ChangeFirstLetterCase();  
  
Please note that, we should still be able to call this extension method using wrapper class style syntax. In fact, behind the scene this is how the method actually gets called. Extension methods are just a syntactic sugar.  
string result = StringHelper.ChangeFirstLetterCase(strName);

So, this means we should also be able to call LINQ extension methods (select, where etc), using wrapper class style syntax. Since all LINQ extension methods are defined in Enumerable class, the syntax will be as shown below.

List<int> Numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

IEnumerable<int> EvenNumbers = Enumerable.Where(Numbers, n => n % 2 == 0);

**LINQ Standard Query Operators**also called as **LINQ extension methods**can be broadly classified into the following categories  
Aggregate Operators  
Grouping Operators  
Restriction Operators  
Projection Operators  
Set Operators  
Partitioning Operators  
Conversion Operators  
Element Operators  
Ordering Operators  
Generation Operators  
Query Execution  
Join Operators  
Custom Sequence Operators  
Quantifiers Operators  
Miscellaneous Operators   
  
   
  
In this video we will discuss the following **LINQ Aggregate**Operators  
Min  
Max  
Sum  
Count  
Average  
Aggregate (**Next Video**)  
  
**Example 1:**

using System;

using System.Linq;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            int[] Numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            int smallestNumber = Numbers.Min();

            int smallestEvenNumber = Numbers.Where(n => n % 2 == 0).Min();

            int largestNumber = Numbers.Max();

            int largestEvenNumber = Numbers.Where(n => n % 2 == 0).Max();

            int sumOfAllNumbers = Numbers.Sum();

            int sumOfAllEvenNumbers = Numbers.Where(n => n % 2 == 0).Sum();

            int countOfAllNumbers = Numbers.Count();

            int countOfAllEvenNumbers = Numbers.Where(n => n % 2 == 0).Count();

            double averageOfAllNumbers = Numbers.Average();

            double averageOfAllEvenNumbers = Numbers.Where(n => n % 2 == 0).Average();

            Console.WriteLine("Smallest Number = " + smallestNumber);

            Console.WriteLine("Smallest Even Number = " + smallestEvenNumber);

            Console.WriteLine("Largest Number = " + largestNumber);

            Console.WriteLine("Largest Even Number = " + largestEvenNumber);

            Console.WriteLine("Sum of All Numbers = " + sumOfAllNumbers);

            Console.WriteLine("Sum of All Even Numbers = " + sumOfAllEvenNumbers);

            Console.WriteLine("Count of All Numbers = " + countOfAllNumbers);

            Console.WriteLine("Count of All Even Numbers = " + countOfAllEvenNumbers);

            Console.WriteLine("Average of All Numbers = " + averageOfAllNumbers);

            Console.WriteLine("Average of All Even Numbers = " + averageOfAllEvenNumbers);

        }

    }

}

**Example 2:**

using System;

using System.Linq;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            string[] countries = { "India", "USA", "UK" };

            int minCount = countries.Min(x => x.Length);

            int maxCount = countries.Max(x => x.Length);

            Console.WriteLine

                   ("The shortest country name has {0} characters in its name", minCount);

            Console.WriteLine

                   ("The longest country name has {0} characters in its name", maxCount);

        }

    }

}

The **WHERE**standard query operator belong to Restriction Operators category in LINQ. Just like SQL, the WHERE standard query operator in LINQ is used to filter rows. The filter expression is specified using a predicate.   
  
   
  
The following are the **2 overloaded versions of WHERE** extension method in Enumerable class

public static IEnumerable<TSource> Where<TSource>(

    this IEnumerable<TSource> source,

    Func<TSource, bool> predicate);

public static IEnumerable<TSource> Where<TSource>(

    this IEnumerable<TSource> source,

    Func<TSource, int, bool> predicate);

**What is a Predicate?**  
A predicate is a function to test each element for a condition  
  
In the following example, the Lambda expression (num => num % 2 == 0) runs for each element in List<int>. If the number is divisible by 2, then a boolean value true is returned otherwise false.

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            IEnumerable<int> evenNumbers = numbers.Where(num => num % 2 == 0);

            foreach (int evenNumber in evenNumbers)

            {

                Console.WriteLine(evenNumber);

            }

        }

    }

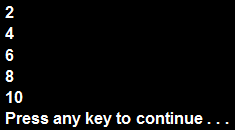
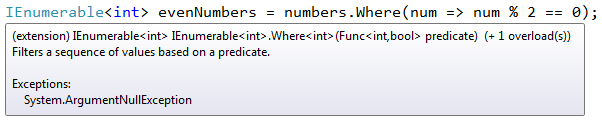
}

// Using SQL like syntax

IEnumerable<int> evenNumbers = from num in numbers

                                                           where num % 2 == 0

                                                           select num;

**Note:**The where query operator is optional.  
  
The program prints all the even numbers   
   
  
When you hover the mouse ove **WHERE**method in the above example, visual studio intellisense shows the following. Notice that in this case, the predicate expects an int input parameter and returns a boolean value. The lambda expression that is passed operates on an int type and should return boolean, otherwise there will be compile time error.   
  
  
**So this means, the line below from the above example**

IEnumerable<int> evenNumbers = numbers.Where(num => num % 2 == 0);

**can be rewritten as shown below**

Func<int, bool> predicate = i => i % 2 == 0;

IEnumerable<int> evenNumbers = numbers.Where(predicate);

**or like below**

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            IEnumerable<int> evenNumbers = numbers.Where(num => IsEven(num));

            foreach (int evenNumber in evenNumbers)

            {

                Console.WriteLine(evenNumber);

            }

        }

        public static bool IsEven(int number)

        {

            if (number % 2 == 0)

            {

                return true;

            }

            else

            {

                return false;

            }

        }

    }

}

**Example 2:**  
The int parameter of the predicate function represents the index of the source element

public static IEnumerable<TSource> Where<TSource>(

    this IEnumerable<TSource> source,

    Func<TSource, int, bool> predicate);

**The following program prints the index position of all the even numbers**

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            List<int> numbers = new List<int> { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

            IEnumerable<int> evenNumberIndexPositions = numbers

                .Select((num, index) => new { Number = num, Index = index })

                .Where(x => x.Number % 2 == 0)

                .Select(x => x.Index);

            foreach (int evenNumber in evenNumberIndexPositions)

            {

                Console.WriteLine(evenNumber);

            }

        }

    }

}

**Example 3:**  
Use the following SQL to create **Departments**and **Employees**tables

Create table Departments

(

     ID int primary key identity,

     Name nvarchar(50),

     Location nvarchar(50)

)

GO

Create table Employees

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50),

     Salary int,

     DepartmentId int foreign key references Departments(Id)

)

GO

Insert into Departments values ('IT', 'New York')

Insert into Departments values ('HR', 'London')

Insert into Departments values ('Payroll', 'Sydney')

GO

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000, 1)

Insert into Employees values ('Steve', 'Pound', 'Male', 45000, 3)

Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000, 1)

Insert into Employees values ('Philip', 'Hastings', 'Male', 45000, 2)

Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000, 2)

Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000, 3)

Insert into Employees values ('John', 'Stanmore', 'Male', 80000, 1)

GO

Add an **ADO.NET entity data model**based on the above 2 tables.  
  
Write a LINQ query to retrieve **IT and HR department names and all the male employees**with in these 2 departments.

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            EmployeeDBContext context = new EmployeeDBContext();

            IEnumerable<Department> departments = context.Departments

                .Where(dept => dept.Name == "IT" || dept.Name == "HR");

            foreach (Department department in departments)

            {

                Console.WriteLine("Department Name = " + department.Name);

                foreach (Employee employee in department

                    .Employees.Where(emp => emp.Gender == "Male"))

                {

                    Console.WriteLine("\tEmployee Name = " + employee.FirstName

                        + " " + employee.LastName);

                }

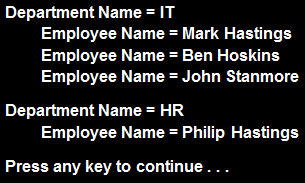
                Console.WriteLine();

            }

        }

    }

}

**Output:**   


Min  
Max  
Sum  
Count  
Average   
  
   
  
Let us understand the use of **Aggregate**() function with examples.   
  
**Example 1:**Consider the following string array.

string[] countries = { "India", "US", "UK", "Canada", "Australia" };

We want to combine all these strings into a single comma separated string. The output of the program should be as shown below.   
India, US, UK, Canada, Australia  
  
**Without LINQ, the program will be as shown below.**

using System;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            string[] countries = { "India", "US", "UK", "Canada", "Australia" };

            string result = string.Empty;

            for (int i = 0; i < countries.Length; i++)

            {

                result = result + countries[i] + ", ";

            }

            int lastIndex = result.LastIndexOf(",");

            result = result.Remove(lastIndex);

            Console.WriteLine(result);

        }

    }

}

**With LINQ Aggregate function**

using System;

using System.Linq;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            string[] countries = { "India", "US", "UK", "Canada", "Australia" };

            string result = countries.Aggregate((a, b) => a + ", " + b);

            Console.WriteLine(result);

        }

    }

}

**How Aggregate() function works?**  
**Step 1.**First **"India"**is concatenated with **"US"**to produce result **"India, US"**  
**Step 2.** Result in **Step 1**is then concatenated with **"UK"**to produce result **"India, US, UK"**

**Step 3:**Result in **Step 2**is then concatenated with **"Canada"**to produce result **"India, US, UK, Canada"**  
  
This goes on until the last element in the array to produce the final single string **"India, US, UK, Canada, Australia"**  
  
**Example 2:** Consider the following integer array

int[] Numbers = { 2, 3, 4, 5 };

**Compute the product of all numbers**  
  
**Without LINQ**

using System;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            int[] Numbers = { 2, 3, 4, 5 };

            int result = 1;

            foreach (int i in Numbers)

            {

                result = result \* i;

            }

            Console.WriteLine(result);

        }

    }

}

**With LINQ:**

using System;

using System.Linq;

namespace Demo

{

    class Program

    {

        static void Main()

        {

            int[] Numbers = { 2, 3, 4, 5 };

            int result = Numbers.Aggregate((a, b) => a \* b);

            Console.WriteLine(result);

        }

    }

}

**How Aggregate() function works?**  
**Step 1:** Multiply **(2X3)**to produce result **6**  
**Step 2:**Result **(6)**in **Step 1**is then multiplied with **4 (6X4)**to produce result **24**  
**Step 3:**Result **(24)**in **Step 2**is then multiplied with **5 (24X5)**to produce final result **120**  
  
**Example 3:**Consider the following integer array

int[] Numbers = { 2, 3, 4, 5 };

One of the overloaded version of **Aggregate()**function has a **Seed**parameter. If we pass **10**as the value for Seed parameter  
int result = Numbers.Aggregate(10, (a, b) => a \* b);  
  
**1200** will be the result   
  
**Step 1:** Multiply **(10X2)**to produce result **20**  
**Step 2:**Result **(20)**in **Step 1**is then multiplied with **3 (20X3)**to produce result **60**  
**Step 3:** Result **(60)**in **Step 2**is then multiplied with **4 (60X4)**to produce result **240**  
**Step 4:**Result **(240)**in **Step 3**is then multiplied with **5 (240X5)**to produce final result **1200**

**Projection Operators (Select & SelectMany)** are used to transform the results of a query. In this video we will discuss **Select**operator and in a later video session we will discuss **SelectMany**operator.   
  
**Select clause**in SQL allows to specify what columns we want to retrieve. In a similar fashion LINQ SELECT standard query operator allows us to specify what properties we want to retrieve. It also allows us to perform calculations.  
  
**For example**, you may have a collection of Employee objects. The following are the properties of the **Employee**class.  
EmployeeID  
FirstName  
LastName  
AnnualSalay  
Gender  
  
**Now using the SELECT projection operator**  
**1.** We can select just **EmployeeID**property OR  
**2.** We can select multiple properties (**FirstName & Gender**) into an anonymous type OR  
**3.** Perform calculations   
    **a)** MonthlySalary = AnnualSalay/12  
    **b)** FullName = FirstName + " " + LastName  
  
We will be using the following **Employee**class for this demo.

public class Employee

{

    public int EmployeeID { get; set; }

    public string FirstName { get; set; }

    public string LastName { get; set; }

    public string Gender { get; set; }

    public int AnnualSalary { get; set; }

    public static List<Employee> GetAllEmployees()

    {

        List<Employee> listEmployees = new List<Employee>

        {

            new Employee

            {

                EmployeeID = 101,

                FirstName = "Tom",

                LastName = "Daely",

                Gender = "Male",

                AnnualSalary = 60000

            },

            new Employee

            {

                EmployeeID = 102,

                FirstName = "Mike",

                LastName = "Mist",

                Gender = "Male",

                AnnualSalary = 72000

            },

            new Employee

            {

                EmployeeID = 103,

                FirstName = "Mary",

                LastName = "Lambeth",

                Gender = "Female",

                AnnualSalary = 48000

            },

            new Employee

            {

                EmployeeID = 104,

                FirstName = "Pam",

                LastName = "Penny",

                Gender = "Female",

                AnnualSalary = 84000

            },

        };

        return listEmployees;

    }

}

**Example 1:** Retrieves just the **EmployeeID**property of all employees

IEnumerable<int> employeeIds = Employee.GetAllEmployees()

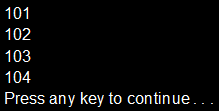
    .Select(emp => emp.EmployeeID);

foreach (int id in employeeIds)

{

    Console.WriteLine(id);

}

**Output:**   
   
  
**Example 2:** Projects **FirstName & Gender**properties of all employees into **anonymous type**.

var result = Employee.GetAllEmployees().Select(emp => new

                    {

                        FirstName = emp.FirstName,

                        Gender = emp.Gender

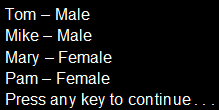
                    });

foreach (var v in result)

{

    Console.WriteLine(v.FirstName + " - " + v.Gender);

}

**Output:**   
   
  
**Example 3:** Computes **FullName and MonthlySalay**of all employees and projects these 2 new computed properties into anonymous type.

var result = Employee.GetAllEmployees().Select(emp => new

{

    FullName = emp.FirstName + " " + emp.LastName,

    MonthlySalary = emp.AnnualSalary / 12

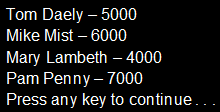
});

foreach (var v in result)

{

    Console.WriteLine(v.FullName + " - " + v.MonthlySalary);

}

**Output:**   
   
  
**Example 4:** Give **10% bonus**to all employees whose annual salary is greater than **50000**and project all such employee's **FirstName, AnnualSalay and Bonus**into anonymous type.

var result = Employee.GetAllEmployees()

                .Where(emp => emp.AnnualSalary > 50000)

                .Select(emp => new

                 {

                    Name = emp.FirstName,

                    Salary = emp.AnnualSalary,

                    Bonus = emp.AnnualSalary \* .1

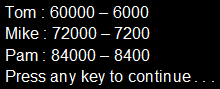
                 });

foreach (var v in result)

{

    Console.WriteLine(v.Name + " : " + v.Salary + " - " + v.Bonus);

}

**Output:**   


**SelectMany**Operator belong to **Projection Operators**category. It is used to project each element of a sequence to an **IEnumerable<T>**and flattens the resulting sequences **into one sequence**.   
  
   
  
Let us understand this with an example. Consider the following **Student**class. **Subjects** property in this class is a **collection of strings**.

public class Student

{

    public string Name { get; set; }

    public string Gender { get; set; }

    public List<string> Subjects { get; set; }

    public static List<Student> GetAllStudetns()

    {

        List<Student> listStudents = new List<Student>

        {

            new Student

            {

                Name = "Tom",

                Gender = "Male",

                Subjects = new List<string> { "ASP.NET", "C#" }

            },

            new Student

            {

                Name = "Mike",

                Gender = "Male",

                Subjects = new List<string> { "ADO.NET", "C#", "AJAX" }

            },

            new Student

            {

                Name = "Pam",

                Gender = "Female",

                Subjects = new List<string> { "WCF", "SQL Server", "C#" }

            },

            new Student

            {

                Name = "Mary",

                Gender = "Female",

                Subjects = new List<string> { "WPF", "LINQ", "ASP.NET" }

            },

        };

        return listStudents;

    }

}

**Example 1:** Projects all subject strings of a given a student to an **IEnumerable<string>**. In this example since we have 4 students, there will be 4 IEnumerable<string> sequences, which are then flattened to form a single sequence i.e a single IEnumerable<string> sequence.

IEnumerable<string> allSubjects = Student.GetAllStudetns().SelectMany(s => s.Subjects);

foreach (string subject in allSubjects)

{

    Console.WriteLine(subject);

}

**Output:**   
   
  
**Example 2:** Rewrite **Example1**using **SQL like syntax.**When using SQL like syntax style, we don't use SelectMany, instead we will have an additional from clause, which will get it's data from the results of the first from clause.

IEnumerable<string> allSubjects = from student in Student.GetAllStudetns()

                                                            from subject in student.Subjects

                                                            select subject;

foreach (string subject in allSubjects)

{

    Console.WriteLine(subject);

}

**Output:**  
Same output as in **Example 1**  
  
**Example 3:**Projects each string to an **IEnumerable<char>**. In this example since we have 2 strings, there will be 2 IEnumerable<char> sequences, which are then flattened to form a single sequence i.e a single IEnumerable<char> sequence.

string[] stringArray =

{

    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",

    "0123456789"

};

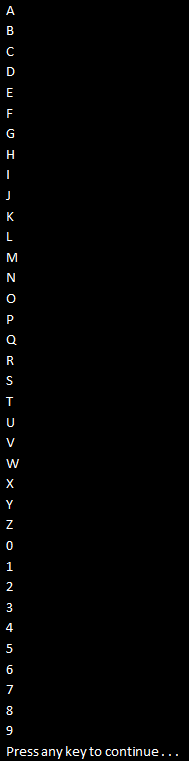
IEnumerable<char> result = stringArray.SelectMany(s => s);

foreach (char c in result)

{

    Console.WriteLine(c);

}

**Output:**   
   
  
**Example 4:** Rewrite **Example3**using **SQL like syntax.**

string[] stringArray =

{

    "ABCDEFGHIJKLMNOPQRSTUVWXYZ",

    "0123456789"

};

IEnumerable<char> result = from s in stringArray

                                                from c in s

                                                select c;

foreach (char c in result)

{

    Console.WriteLine(c);

}

**Output:**  
Same output as in **Example 3**  
  
**Example 5:**Selects only the distinct subjects

IEnumerable<string> allSubjects = Student.GetAllStudetns()

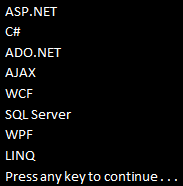
                                                                   .SelectMany(s => s.Subjects).Distinct();

foreach (string subject in allSubjects)

{

    Console.WriteLine(subject);

}

**Output:**   
   
  
**Example 6:** Rewrite **Example 5**using **SQL like syntax.**

IEnumerable<string> allSubjects = (from student in Student.GetAllStudetns()

                                                             from subject in student.Subjects

                                                             select subject).Distinct();

foreach (string subject in allSubjects)

{

    Console.WriteLine(subject);

}

**Output:**  
Same output as in **Example 5**  
  
**Example 7:**Selects student name along with all the subjects

var result = Student.GetAllStudetns().SelectMany(s => s.Subjects, (student, subject) =>

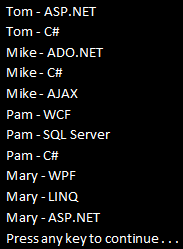
    new { StudentName = student.Name, Subject = subject });

foreach (var v in result)

{

    Console.WriteLine(v.StudentName + " - " + v.Subject);

}

**Output:**   
   
  
**Example 8:** Rewrite **Example 7**using **SQL like syntax.**

var result = from student in Student.GetAllStudetns()

                    from subject in student.Subjects

                    select new { StudnetName = student.Name, Subject = subject };

foreach (var v in result)

{

    Console.WriteLine(v.StudnetName + " - " + v.Subject);

}

**Output:**  
Same output as in **Example 7**

Let us understand the **difference between Select and SelectMany**with an example.   
  
   
  
We will be using the following **Student**class in this demo. **Subjects**property in this class is a collection of strings.

public class Student

{

    public string Name { get; set; }

    public string Gender { get; set; }

    public List<string> Subjects { get; set; }

    public static List<Student> GetAllStudetns()

    {

        List<Student> listStudents = new List<Student>

        {

            new Student

            {

                Name = "Tom",

                Gender = "Male",

                Subjects = new List<string> { "ASP.NET", "C#" }

            },

            new Student

            {

                Name = "Mike",

                Gender = "Male",

                Subjects = new List<string> { "ADO.NET", "C#", "AJAX" }

            },

            new Student

            {

                Name = "Pam",

                Gender = "Female",

                Subjects = new List<string> { "WCF", "SQL Server", "C#" }

            },

            new Student

            {

                Name = "Mary",

                Gender = "Female",

                Subjects = new List<string> { "WPF", "LINQ", "ASP.NET" }

            },

        };

        return listStudents;

    }

}

In this example, the **Select()**method returns **List of List<string>**. To print all the subjects we will have to use **2 nested foreach loops**.

IEnumerable<List<string>> result = Student.GetAllStudetns().Select(s => s.Subjects);

foreach (List<string> stringList in result)

{

    foreach (string str in stringList)

    {

        Console.WriteLine(str);

    }

}

**SelectMany()** on the other hand, flattens queries that return lists of lists into a **single list.**So in this case to print all the subjects we have to use just one foreach loop.

IEnumerable<string> result = Student.GetAllStudetns().SelectMany(s => s.Subjects);

foreach (string str in result)

{

    Console.WriteLine(str);

}

**Output:**   


**Part 10 - Ordering Operators in LINQ**

**Suggested Videos**  
[Part 7 - Projection Operators](http://csharp-video-tutorials.blogspot.com/2014/07/part-7-projection-operators-in-linq_9.html)  
[Part 8 - SelectMany Operator](http://csharp-video-tutorials.blogspot.com/2014/07/part-8-selectmany-operator-in-linq.html)  
[Part 9 - Difference between Select and SelectMany](http://csharp-video-tutorials.blogspot.com/2014/07/part-9-difference-between-select-and.html)   
  
   
  
The following **5 standard LINQ query operators belong to Ordering Operators**category  
OrderBy  
OrderByDescending  
ThenBy  
ThenByDescending  
Reverse   
  
   
  
**OrderBy, OrderByDescending, ThenBy,** and **ThenByDescending**can be used to sort data. **Reverse**method simply reverses the items in a given collection.  
  
We will use the following **Student**class in this demo.

public class Student

{

    public int StudentID { get; set; }

    public string Name { get; set; }

    public int TotalMarks { get; set; }

    public static List<Student> GetAllStudents()

    {

        List<Student> listStudents = new List<Student>

        {

            new Student

            {

                StudentID= 101,

                Name = "Tom",

                TotalMarks = 800

            },

            new Student

            {

                StudentID= 102,

                Name = "Mary",

                TotalMarks = 900

            },

            new Student

            {

                StudentID= 103,

                Name = "Valarie",

                TotalMarks = 800

            },

            new Student

            {

                StudentID= 104,

                Name = "John",

                TotalMarks = 800

            },

        };

        return listStudents;

    }

}

**Example 1:** Sort **Students**by **Name**in **ascending**order

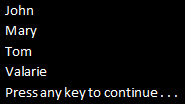
IEnumerable<Student> result = Student.GetAllStudents().OrderBy(s => s.Name);

foreach (Student student in result)

{

    Console.WriteLine(student.Name);

}

**Output:**   
   
  
**Example 2:** Rewrite **Example 1**using **SQL like**syntax

IEnumerable<Student> result = from student in Student.GetAllStudents()

                                                      orderby student.Name

                                                      select student;

foreach (Student student in result)

{

    Console.WriteLine(student.Name);

}

**Output:**  
Same as in **Example 1**  
  
**Example 3:**Sort **Students**by **Name**in **descending**order

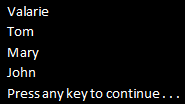
IEnumerable<Student> result = Student.GetAllStudents().OrderByDescending(s => s.Name);

foreach (Student student in result)

{

    Console.WriteLine(student.Name);

}

**Output:**   
   
  
**Example 4:** Rewrite **Example 3** using **SQL like**syntax

IEnumerable<Student> result = from student in Student.GetAllStudents()

                                                      orderby student.Name descending

                                                      select student;

foreach (Student student in result)

{

    Console.WriteLine(student.Name);

}

**Output:**  
Same as in **Example 1**

**Part 11 - Ordering Operators in LINQ - II**

**Suggested Videos**  
[Part 8 - SelectMany Operator](http://csharp-video-tutorials.blogspot.com/2014/07/part-8-selectmany-operator-in-linq.html)  
[Part 9 - Difference between Select and SelectMany](http://csharp-video-tutorials.blogspot.com/2014/07/part-9-difference-between-select-and.html)   
[Part 10 - Ordering Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-10-ordering-operators-in-linq_13.html)   
  
   
  
This is continuation to [Part 10](http://csharp-video-tutorials.blogspot.com/2014/07/part-10-ordering-operators-in-linq_13.html). Please watch [Part 10](http://csharp-video-tutorials.blogspot.com/2014/07/part-10-ordering-operators-in-linq_13.html) before proceeding.   
  
   
  
**The following 5 standard LINQ query operators belong to Ordering Operators category**  
OrderBy  
OrderByDescending  
ThenBy  
ThenByDescending  
Reverse  
  
In [Part 10](http://csharp-video-tutorials.blogspot.com/2014/07/part-10-ordering-operators-in-linq_13.html), we discussed **OrderBy**& **OrderByDescending**operators. In this video we will discuss  
ThenBy  
ThenByDescending  
Reverse  
  
OrderBy, OrderByDescending, ThenBy, and ThenByDescending can be used to sort data. Reverse method simply reverses the items in a given collection.  
  
We will use the following **Student**class in this demo.

public class Student

{

    public int StudentID { get; set; }

    public string Name { get; set; }

    public int TotalMarks { get; set; }

    public static List<Student> GetAllStudetns()

    {

        List<Student> listStudents = new List<Student>

        {

            new Student

            {

                StudentID= 101,

                Name = "Tom",

                TotalMarks = 800

            },

            new Student

            {

                StudentID= 102,

                Name = "Mary",

                TotalMarks = 900

            },

            new Student

            {

                StudentID= 103,

                Name = "Pam",

                TotalMarks = 800

            },

            new Student

            {

                StudentID= 104,

                Name = "John",

                TotalMarks = 800

            },

            new Student

            {

                StudentID= 105,

                Name = "John",

                TotalMarks = 800

            },

        };

        return listStudents;

    }

}

**OrderBy**or **OrderByDescending**work fine when we want to sort a collection just by one value or expression.   
  
If want to sort by more than one value or expression, that's when we use **ThenBy**or **ThenByDescending**along with **OrderBy**or **OrderByDescending.**

**OrderBy**or **OrderByDescending**performs the primary sort. **ThenBy**or **ThenByDescending**is used for adding secondary sort. Secondary Sort operators (**ThenBy**or **ThenByDescending**) can be used more than once in the same LINQ query.  
  
**Example 1:**  
**a)** Sorts **Students**first by **TotalMarks**in ascending order(Primary Sort)   
**b)** The 4 Students with **TotalMarks**of **800,**will then be sorted by Name in ascending order (First Secondary Sort)  
**c)** The **2 Students**with **Name**of **John**, will then be sorted by **StudentID**in ascending order (Second Secondary Sort)

IEnumerable<Student> result = Student.GetAllStudetns()

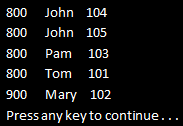
    .OrderBy(s => s.TotalMarks).ThenBy(s => s.Name).ThenBy(s => s.StudentID);

foreach (Student student in result)

{

    Console.WriteLine(student.TotalMarks + "\t" + student.Name + "\t" + student.StudentID);

}

**Output:**   
   
  
**Example 2:** Rewrite **Example 1**using **SQL**like syntax. With SQL like syntax we donot use **ThenBy**or **ThenByDescending,**instead we specify the sort expressions using a comma separated list. The first sort expression will be used for primary sort and the subsequent sort expressions for secondary sort.

IEnumerable<Student> result = from student in Student.GetAllStudetns()

                                                      orderby student.TotalMarks, student.Name, student.StudentID

                                                      select student;

foreach (Student student in result)

{

    Console.WriteLine(student.TotalMarks + "\t" + student.Name + "\t" + student.StudentID);

}

**Example 3:** Reverses the items in the collection.

IEnumerable<Student> students = Student.GetAllStudetns();

Console.WriteLine("Before calling Reverse");

foreach (Student s in students)

{

    Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);

}

Console.WriteLine();

IEnumerable<Student> result = students.Reverse();

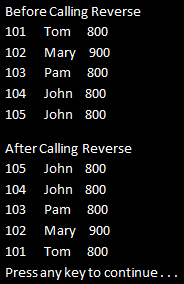
Console.WriteLine("After calling Reverse");

foreach (Student s in result)

{

    Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);

}

**Output:**   


**Part 12 - Partitioning Operators in LINQ**

**Suggested Videos**  
[Part 9 - Difference between Select and SelectMany](http://csharp-video-tutorials.blogspot.com/2014/07/part-9-difference-between-select-and.html)  
[Part 10 - Ordering Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-10-ordering-operators-in-linq_13.html)   
[Part 11 - Ordering Operators in LINQ - II](http://csharp-video-tutorials.blogspot.com/2014/07/part-11-ordering-operators-in-linq-ii.html)   
  
   
  
**The following 4 standard LINQ query operators belong to Partitioning Operators category**  
Take  
Skip  
TakeWhile  
SkipWhile   
  
   
  
**Take**method returns a specified number of elements from the start of the collection. The number of items to return is specified using the count parameter this method expects.  
  
**Skip**method skips a specified number of elements in a collection and then returns the remaining elements. The number of items to skip is specified using the count parameter this method expects.   
  
**Please Note:** For the same argument value, the Skip method returns all of the items that the Take method would not return.  
  
**TakeWhile**method returns elements from a collection as long as the given condition specified by the predicate is true.   
  
**SkipWhile**method skips elements in a collection as long as the given condition specified by the predicate is true, and then returns the remaining elements.  
  
**Example 1:**Retrieves only the first 3 countries of the array.

string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

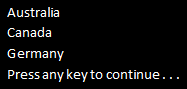
IEnumerable<string> result = countries.Take(3);

foreach (string country in result)

{

    Console.WriteLine(country);

}

**Output:**   
   
  
**Example 2:** Rewrite **Example 1**using SQL like syntax

string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

IEnumerable<string> result = (from country in countries

                                                   select country).Take(3);

foreach (string country in result)

{

    Console.WriteLine(country);

}

**Example 3:** Skips the first 3 countries and retrieves the rest of them

string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

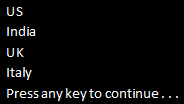
IEnumerable<string> result = countries.Skip(3);

foreach (string country in result)

{

    Console.WriteLine(country);

}

**Output:**   
   
  
**Example 4:** Return countries starting from the beginning of the array until a country name is hit that does not have length greater than 2 characters.

string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

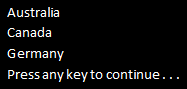
IEnumerable<string> result = countries.TakeWhile(s => s.Length > 2);

foreach (string country in result)

{

    Console.WriteLine(country);

}

**Output:**   
   
  
**Example 5:** Skip elements starting from the beginning of the array, until a country name is hit that does not have length greater than 2 characters, and then return the remaining elements.

string[] countries = { "Australia", "Canada", "Germany", "US", "India", "UK", "Italy" };

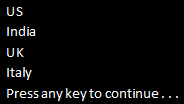
IEnumerable<string> result = countries.SkipWhile(s => s.Length > 2);

foreach (string country in result)

{

    Console.WriteLine(country);

}

**Output:**   


**Part 13 - Implement paging using skip and take operators**

**Suggested Videos**  
[Part 10 - Ordering Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-10-ordering-operators-in-linq_13.html)  
[Part 11 - Ordering Operators in LINQ - II](http://csharp-video-tutorials.blogspot.com/2014/07/part-11-ordering-operators-in-linq-ii.html)   
[Part 12 - Partitioning Operators](http://csharp-video-tutorials.blogspot.com/2014/07/part-12-partitioning-operators-in-linq.html)   
  
   
  
In this video, we will discuss **implementing paging using Skip and Take**operators in LINQ.   
  
   
  
We will use the following Student class in this demo. Notice that, there are **11 total Students**. We want to display a maximum of **3 students per page**. So there will be **4 total pages.**The last page, i.e **Page 4**will display the last **2 students**.

public class Student

{

    public int StudentID { get; set; }

    public string Name { get; set; }

    public int TotalMarks { get; set; }

    public static List<Student> GetAllStudetns()

    {

        List<Student> listStudents = new List<Student>

        {

            new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

            new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

            new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 },

            new Student { StudentID= 104, Name = "John", TotalMarks = 800 },

            new Student { StudentID= 105, Name = "John", TotalMarks = 800 },

            new Student { StudentID= 106, Name = "Brian", TotalMarks = 700 },

            new Student { StudentID= 107, Name = "Jade", TotalMarks = 750 },

            new Student { StudentID= 108, Name = "Ron", TotalMarks = 850 },

            new Student { StudentID= 109, Name = "Rob", TotalMarks = 950 },

            new Student { StudentID= 110, Name = "Alex", TotalMarks = 750 },

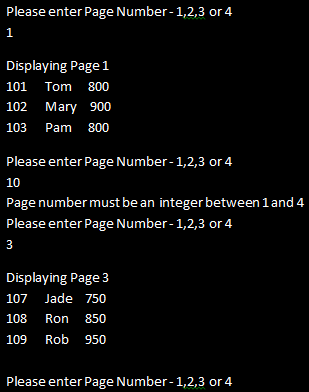
            new Student { StudentID= 111, Name = "Susan", TotalMarks = 860 },

        };

        return listStudents;

    }

}

**Here is what we want to do**  
**1.** The program should prompt the user to enter a page number. The Page number must be between 1 and 4.  
**2.** If the user does not enter a valid page number, the program should prompt the user to enter a valid page number.  
**3.** Once a valid page number is entered, the program should display the correct set of Students  
  
For example, **the output of the program**should be as shown below.  
   
  
The following console application use **Skip()** and **Take()**operators to achieve this.

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        public static void Main()

        {

            IEnumerable<Student> students = Student.GetAllStudetns();

            do

            {

                Console.WriteLine("Please enter Page Number - 1,2,3 or 4");

                int pageNumber = 0;

                if (int.TryParse(Console.ReadLine(), out pageNumber))

                {

                    if (pageNumber >= 1 && pageNumber <= 4)

                    {

                        int pageSize = 3;

                        IEnumerable<Student> result = students

                                                     .Skip((pageNumber - 1) \* pageSize).Take(pageSize);

                        Console.WriteLine();

                        Console.WriteLine("Displaying Page " + pageNumber);

                        foreach (Student student in result)

                        {

                            Console.WriteLine(student.StudentID + "\t" +

                                                                        student.Name + "\t" + student.TotalMarks);

                        }

                        Console.WriteLine();

                    }

                    else

                    {

                        Console.WriteLine("Page number must be an integer between 1 and 4");

                    }

                }

                else

                {

                    Console.WriteLine("Page number must be an integer between 1 and 4");

                }

            } while (1 == 1);

        }

    }

}

**Please Note:** The condition in the while loop puts the program in an infinite loop. To end the program, simply close the console window.

**Part 14 - LINQ query deferred execution**

**Suggested Videos**  
[Part 11 - Ordering Operators in LINQ - II](http://csharp-video-tutorials.blogspot.com/2014/07/part-11-ordering-operators-in-linq-ii.html)  
[Part 12 - Partitioning Operators](http://csharp-video-tutorials.blogspot.com/2014/07/part-12-partitioning-operators-in-linq.html)   
[Part 13 - Implement paging using skip and take operators](http://csharp-video-tutorials.blogspot.com/2014/07/part-13-implement-paging-using-skip-and.html)   
  
   
  
In this video we will discuss the concept of **deferred execution**. LINQ queries have two different behaviors of execution  
**1.** Deferred execution  
**2.** Immediate execution   
  
   
  
**LINQ operators can be broadly classified into 2 categories based on the behaviour of query execution**  
**1. Deferred or Lazy Operators -**These query operators use deferred execution.   
Examples - select, where, Take, Skip etc  
**2. Immediate or Greedy Operators -**These query operators use immediate execution.   
Examples - count, average, min, max, ToList etc  
  
Let us understand these 2 behaviors with examples.   
  
**LINQ Deferred Execution Example**

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    public class Student

    {

        public int StudentID { get; set; }

        public string Name { get; set; }

        public int TotalMarks { get; set; }

    }

    class Program

    {

        public static void Main()

        {

            List<Student> listStudents = new List<Student>

            {

                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }

            };

            // LINQ Query is only defined here and is not executed at this point

            // If the query is executed at this point, the result should not display Tim

            IEnumerable<Student> result = from student in listStudents

                                          where student.TotalMarks == 800

                                          select student;

            // Add a new student object with TotalMarks = 800 to the source

            listStudents.Add(new Student { StudentID = 104, Name = "Tim", TotalMarks = 800 });

            // The above query is actually executed when we iterate thru the sequence

            // using the foreach loop. This is proved as Tim is also included in the result

            foreach (Student s in result)

            {

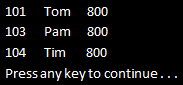
                Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);

            }

        }

    }

}

**Output:**   
   
  
**LINQ Immediate Execution Example 1**

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    public class Student

    {

        public int StudentID { get; set; }

        public string Name { get; set; }

        public int TotalMarks { get; set; }

    }

    class Program

    {

        public static void Main()

        {

            List<Student> listStudents = new List<Student>

            {

                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }

            };

            // Since we are using ToList() which is a greedy operator

            // the LINQ Query is executed immediately at this point

            IEnumerable<Student> result = (from student in listStudents

                                           where student.TotalMarks == 800

                                           select student).ToList();

            // Adding a new student object with TotalMarks = 800 to the source

            // will have no effect on the result as the query is already executed

            listStudents.Add(new Student { StudentID = 104, Name = "Tim", TotalMarks = 800 });

            // The above query is executed at the point where it is defined.

            // This is proved as Tim is not included in the result

            foreach (Student s in result)

            {

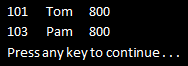
                Console.WriteLine(s.StudentID + "\t" + s.Name + "\t" + s.TotalMarks);

            }

        }

    }

}

**Output:**   
   
  
**LINQ Immediate Execution Example 2**

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    public class Student

    {

        public int StudentID { get; set; }

        public string Name { get; set; }

        public int TotalMarks { get; set; }

    }

    class Program

    {

        public static void Main()

        {

            List<Student> listStudents = new List<Student>

            {

                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }

            };

            // Since we are using Count() operator, the LINQ Query is executed at this point

            int result = (from student in listStudents

                          where student.TotalMarks == 800

                          select student).Count();

            // Adding a new student object with TotalMarks = 800 to the source

            // will have no effect on the result as the query is already executed

            listStudents.Add(new Student { StudentID = 104, Name = "Tim", TotalMarks = 800 });

            // The above query is executed at the point where it is defined.

            // This is proved as Tim is not included in the count

            Console.WriteLine("Students with Total Marks = 800 : " + result);

        }

    }

}

**Output:**   
force linq query execute immediately

**Part 15 - Conversion Operators in LINQ**

**Suggested Videos**  
[Part 12 - Partitioning Operators](http://csharp-video-tutorials.blogspot.com/2014/07/part-12-partitioning-operators-in-linq.html)  
[Part 13 - Implement paging using skip and take operators](http://csharp-video-tutorials.blogspot.com/2014/07/part-13-implement-paging-using-skip-and.html)   
[Part 14 - LINQ query deferred execution](http://csharp-video-tutorials.blogspot.com/2014/07/part-14-linq-query-deferred-execution.html)   
  
   
  
The following standard LINQ query operators belong to **Conversion Operators**category  
ToList  
ToArray  
ToDictionary  
ToLookup  
Cast  
OfType  
AsEnumerable   
AsQueryable   
  
   
  
**ToList operator** extracts all of the items from the source sequence and returns a new **List<T>**. This operator causes the query to be executed immediately. This operator does not use deferred execution.  
  
**Example 1:**Convert int array to List<int>

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        public static void Main()

        {

            int[] numbers = { 1, 2, 3, 4, 5 };

            List<int> result = numbers.ToList();

            foreach (int i in result)

            {

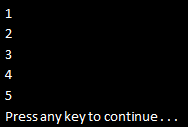
                Console.WriteLine(i);

            }

        }

    }

}

**Output:**   
   
  
**ToArray** operator extracts all of the items from the source sequence and returns a new Array. This operator causes the query to be executed immediately. This operator does not use deferred execution.  
  
**Example 2:**Convert List<string> to string array. The items in the array should be sorted in ascending order.

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        public static void Main()

        {

            List<string> countries = new List<string> { "US", "India", "UK", "Australia", "Canada" };

            string[] result = (from country in countries

                               orderby country ascending

                               select country).ToArray();

            foreach (string str in result)

            {

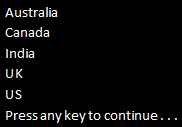
                Console.WriteLine(str);

            }

        }

    }

}

**Output:**   
   
  
**ToDictionary** operator extracts all of the items from the source sequence and returns a new Dictionary. This operator causes the query to be executed immediately. This operator does not use deferred execution.  
  
**Example 3 :**Convert List<Student> to a Dictionary. StudentID should be the key and Name should be the value. In this example, we are using the overloaded of ToDictionary() that takes 2 parameters

**a) keySelector** - A function to extract a key from each element  
**b) elementSelector** - A function to produce a result element from each element in the sequence

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    public class Student

    {

        public int StudentID { get; set; }

        public string Name { get; set; }

        public int TotalMarks { get; set; }

    }

    class Program

    {

        public static void Main()

        {

            List<Student> listStudents = new List<Student>

            {

                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }

            };

            Dictionary<int, string> result = listStudents

                                                                     .ToDictionary(x => x.StudentID, x => x.Name);

            foreach (KeyValuePair<int, string> kvp in result)

            {

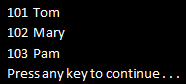
                Console.WriteLine(kvp.Key + " " + kvp.Value);

            }

        }

    }

}

**Output:**   
   
  
**Example 4 :** Convert List<Student> to a Dictionary. StudentID should be the key and Student object should be the value. In this example, we are using the overloaded of ToDictionary() that takes 1 parameter  
**a) keySelector** - A function to extract a key from each element

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    public class Student

    {

        public int StudentID { get; set; }

        public string Name { get; set; }

        public int TotalMarks { get; set; }

    }

    class Program

    {

        public static void Main()

        {

            List<Student> listStudents = new List<Student>

            {

                new Student { StudentID= 101, Name = "Tom", TotalMarks = 800 },

                new Student { StudentID= 102, Name = "Mary", TotalMarks = 900 },

                new Student { StudentID= 103, Name = "Pam", TotalMarks = 800 }

            };

            Dictionary<int, Student> result = listStudents.ToDictionary(x => x.StudentID);

            foreach (KeyValuePair<int, Student> kvp in result)

            {

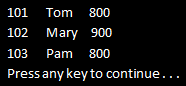
                Console.WriteLine(kvp.Key + "\t" + kvp.Value.Name + "\t" + kvp.Value.TotalMarks);

            }

        }

    }

}

**Output:**   
   
  
**Please Note:** Keys in the dictionary must be unique. If two identical keys are created by the keySelector function, the following System.ArgumentException will be thrown at runtime.  
Unhandled Exception: System.ArgumentException: An item with the same key has already been added.  
  
**ToLookup** creates a Lookup. Just like a dictionary, a Lookup is a collection of key/value pairs. A dictionary cannot contain keys with identical values, where as a Lookup can.  
  
**Example 5:** Create 2 Lookups. First lookup should group Employees by JobTitle, and second lookup should group Employees by City.

using System;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    public class Employee

    {

        public string Name { get; set; }

        public string JobTitle { get; set; }

        public string City { get; set; }

    }

    class Program

    {

        public static void Main()

        {

            List<Employee> listEmployees = new List<Employee>

            {

                new Employee() { Name = "Ben", JobTitle = "Developer", City = "London" },

                new Employee() { Name = "John", JobTitle = "Sr. Developer", City ="Bangalore" },

                new Employee() { Name = "Steve", JobTitle = "Developer", City = "Bangalore"},

                new Employee() { Name = "Stuart", JobTitle = "Sr. Developer", City ="London" },

                new Employee() { Name = "Sara", JobTitle = "Developer", City = "London" },

                new Employee() { Name = "Pam", JobTitle = "Developer", City = "London" }

            };

            // Group employees by JobTitle

            var employeesByJobTitle = listEmployees.ToLookup(x => x.JobTitle);

            Console.WriteLine("Employees Grouped By JobTitle");

            foreach (var kvp in employeesByJobTitle)

            {

                Console.WriteLine(kvp.Key);

                // Lookup employees by JobTitle

                foreach (var item in employeesByJobTitle[kvp.Key])

                {

                    Console.WriteLine("\t" + item.Name + "\t" + item.JobTitle + "\t" + item.City);

                }

            }

            Console.WriteLine(); Console.WriteLine();

            // Group employees by City

            var employeesByCity = listEmployees.ToLookup(x => x.City);

            Console.WriteLine("Employees Grouped By City");

            foreach (var kvp in employeesByCity)

            {

                Console.WriteLine(kvp.Key);

                // Lookup employees by City

                foreach (var item in employeesByCity[kvp.Key])

                {

                    Console.WriteLine("\t" + item.Name + "\t" + item.JobTitle + "\t" + item.City);

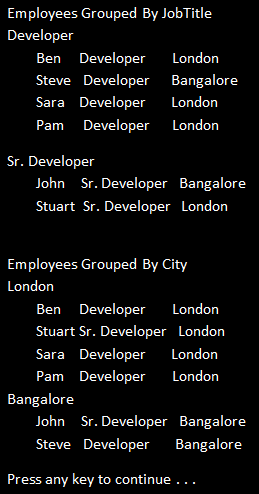
                }

            }

        }

    }

}

Output:   


**Part 16 - Cast and OfType operators in LINQ**

**Suggested Videos**  
[Part 13 - Implement paging using skip and take operators](http://csharp-video-tutorials.blogspot.com/2014/07/part-13-implement-paging-using-skip-and.html)  
[Part 14 - LINQ query deferred execution](http://csharp-video-tutorials.blogspot.com/2014/07/part-14-linq-query-deferred-execution.html)   
[Part 15 - Conversion Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-15-conversion-operators-in-linq.html)   
  
   
  
The following standard LINQ query operators belong to **Conversion Operators**category  
ToList  
ToArray  
ToDictionary  
ToLookup  
Cast  
OfType  
AsEnumerable   
AsQueryable   
  
   
  
We discussed the following operators in [Part 15](http://csharp-video-tutorials.blogspot.com/2014/07/part-15-conversion-operators-in-linq.html)  
ToList  
ToArray  
ToDictionary  
ToLookup  
  
**In this video we will discuss**  
**1.** Cast and OfType operators  
**2.** Difference between Cast and OfType operators  
**3.** When to use one over the other  
  
**Cast operator**attempts to convert all of the items within an existing collection to another type and return them in a new collection. If an item fails conversion an exception will be thrown. This method uses deferred execution.  
  
**Example :**

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        public static void Main()

        {

            ArrayList list = new ArrayList();

            list.Add(1);

            list.Add(2);

            list.Add(3);

            // The following item causes an exception

            // list.Add("ABC");

            IEnumerable<int> result = list.Cast<int>();

            foreach (int i in result)

            {

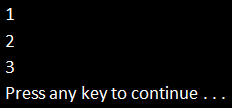
                Console.WriteLine(i);

            }

        }

    }

}

**Output :**   
   
  
**OfType operator**will return only elements of the specified type. The other type elements are simply ignored and excluded from the result set.  
  
**Example :** In the example below, items **"4"**and **"ABC"**will be ignored from the result set. No exception will be thrown.

using System;

using System.Collections;

using System.Collections.Generic;

using System.Linq;

namespace Demo

{

    class Program

    {

        public static void Main()

        {

            ArrayList list = new ArrayList();

            list.Add(1);

            list.Add(2);

            list.Add(3);

            list.Add("4");

            list.Add("ABC");

            IEnumerable<int> result = list.OfType<int>();

            foreach (int i in result)

            {

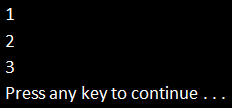
                Console.WriteLine(i);

            }

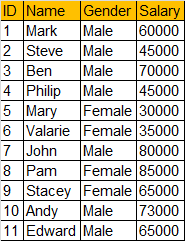
        }

    }

}

**Output** :   
   
  
**What is the difference between Cast and OfType operators**  
OfType operator returns only the elements of the specified type and the rest of the items in the collection will be ignored and excluded from the result.   
  
Cast operator will try to cast all the elements in the collection into the specified type. If some of the items fail conversion, InvalidCastException will be thrown.  
  
**When to use Cast over OfType and vice versa?**  
We would generally use Cast when the following 2 conditions are met  
**1.** We want to cast all the items in the collection &  
**2.** We know for sure the collection contains only elements of the specified type  
  
If we want to filter the elements and return only the ones of the specified type, then we would use OfType.

**Part 17 - AsEnumerable and AsQueryable in LINQ**

**Suggested Videos**  
[Part 14 - LINQ query deferred execution](http://csharp-video-tutorials.blogspot.com/2014/07/part-14-linq-query-deferred-execution.html)  
[Part 15 - Conversion Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-15-conversion-operators-in-linq.html)   
[Part 16 - Cast and OfType operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-16-cast-and-oftype-operators-in.html)   
  
   
  
In this video we will discuss the use of **AsEnumerable**and **AsQueryable**operators in LINQ. Both of these operators belong to **Conversion Operators**category.   
  
   
  
**AsQueryable operator:** There are 2 overloaded versions of this method.   
  
One overloaded version converts System.Collections.IEnumerable to System.Linq.IQueryable  
  
The other overloaded version converts a generic System.Collections.Generic.IEnumerable<T> to a generic System.Linq.IQueryable<T>  
  
The main use of **AsQueryable**operator is unit testing to mock a **queryable**data source using an in-memory data source. We will discuss this operator in detail with examples in unit testing video series.  
  
**AsEnumerable operator:** Let us understand the use of this operator with an example. We will be using the following **Employees**table in this demo.   
   
  
**Step 1:** Execute the following SQL Script to create and populate **Employees**Table

Create Table Employees

(

     ID int primary key identity,

     Name nvarchar(50),

     Gender nvarchar(50),

     Salary int

)

GO

Insert into Employees Values('Mark','Male','60000')

Insert into Employees Values('Steve','Male','45000')

Insert into Employees Values('Ben','Male','70000')

Insert into Employees Values('Philip','Male','45000')

Insert into Employees Values('Mary','Female','30000')

Insert into Employees Values('Valarie','Female','35000')

Insert into Employees Values('John','Male','80000')

Insert into Employees Values('Pam','Female','85000')

Insert into Employees Values('Stacey','Female','65000')

Insert into Employees Values('Andy','Male','73000')

Insert into Employees Values('Edward','Male','65000')

GO

**Step 2:** Create a new Console Application. Name it **Demo**.  
  
**Step 3:**Right click on the Demo project in Solution Explorer and Add a new LINQ to SQL Classes. Name it **EmployeeDB.dbml**.  
  
**Step 4:**Click on **View**menu, and select **"Server Explorer".**  Expand **Data Connections**and then Drag and Drop **Employees**table onto **EmployeeDB.dbml**designer surface.  
  
**Step 5:**Copy and paste the following code in Program.cs file. The linq query in this sample, retrieves the **TOP 5 Male Employees By Salary**.

using System;

using System.Linq;

namespace Demo

{

    class Program

    {

        public static void Main()

        {

            EmployeeDBDataContext dbContext = new EmployeeDBDataContext();

            // TOP 5 Male Employees By Salary

            var result = dbContext.Employees.Where(x => x.Gender == "Male")

                                    .OrderByDescending(x => x.Salary).Take(5);

            Console.WriteLine("Top 5 Salaried Male Employees");

            foreach (Employee e in result)

            {

                Console.WriteLine(e.Name + "\t" + e.Gender + "\t" + e.Salary);

            }

        }

    }

}

**Step 6:** Now open **SQL Profiler**and run a new trace and then run the console application.  
  
**Step 7:**Notice that the following SQL Query is executed against the database.

exec sp\_executesql N'SELECT TOP (5) [t0].[ID], [t0].[Name], [t0].[Gender], [t0].[Salary]

FROM [dbo].[Employees] AS [t0]

WHERE [t0].[Gender] = @p0

ORDER BY [t0].[Salary] DESC',N'@p0 nvarchar(4000)',@p0=N'Male'

**Step 8:** Change the LINQ query in the console application   
  
**FROM**

varresult=dbContext.Employees.Where**(**x=>x.Gender=="Male"**)**

.OrderByDescending**(**x=>x.Salary**)**.Take**(**5**);**  
  
**TO**

varresult=dbContext.Employees.AsEnumerable**()**

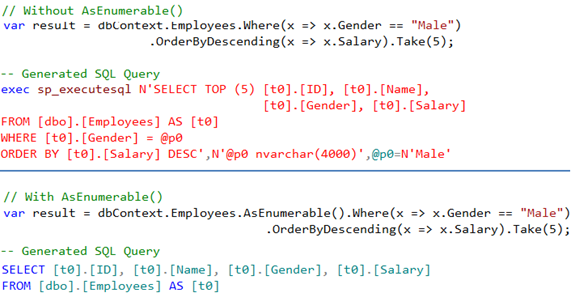
.Where**(**x=>x.Gender=="Male"**)**

.OrderByDescending**(**x=>x.Salary**)**.Take**(**5**);**

**Step 9:**Run the console application and notice the query generated in SQL Profiler.

SELECT [t0].[ID], [t0].[Name], [t0].[Gender], [t0].[Salary]

FROM [dbo].[Employees] AS [t0]

**Summary:**   
   
  
**AsEnumerable operator breaks the query into 2 parts**  
**1.** The "inside part" that is the query before AsEnumerable operator is executed as Linq-to-SQL  
**2.** The "ouside part" that is the query after AsEnumerable operator is executed as Linq-to-Objects  
  
So in this example the following SQL Query is executed against SQL Server, all the data is brought into the console application and then the WHERE, ORDERBY & TOP operators are applied on the client-side

SELECT [t0].[ID], [t0].[Name], [t0].[Gender], [t0].[Salary]

FROM [dbo].[Employees] AS [t0]

So in short, use **AsEnumerable** operator to move query processing to the client side.

**Part 18 - GroupBy in LINQ**

**Suggested Videos**  
[Part 15 - Conversion Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-15-conversion-operators-in-linq.html)  
[Part 16 - Cast and OfType operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-16-cast-and-oftype-operators-in.html)   
[Part 17 - AsEnumerable and AsQueryable in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-17-asenumerable-and-asqueryable-in.html)   
  
   
  
**GroupBy** operator belong to **Grouping Operators**category. This operator takes a flat sequence of items, organize that sequence into groups (**IGrouping<K,V>**) based on a specific key and return groups of sequences.    
  
   
  
In short, **GroupBy**creates and returns a sequence of **IGrouping<K,V>**  
  
Let us understand GroupBy with examples.  
  
We will use the following **Employee**class in this demo

public class Employee

{

    public int ID { get; set; }

    public string Name { get; set; }

    public string Gender { get; set; }

    public string Department { get; set; }

    public int Salary { get; set; }

    public static List<Employee> GetAllEmployees()

    {

        return new List<Employee>()

        {

            new Employee { ID = 1, Name = "Mark", Gender = "Male",

                                         Department = "IT", Salary = 45000 },

            new Employee { ID = 2, Name = "Steve", Gender = "Male",

                                         Department = "HR", Salary = 55000 },

            new Employee { ID = 3, Name = "Ben", Gender = "Male",

                                         Department = "IT", Salary = 65000 },

            new Employee { ID = 4, Name = "Philip", Gender = "Male",

                                         Department = "IT", Salary = 55000 },

            new Employee { ID = 5, Name = "Mary", Gender = "Female",

                                         Department = "HR", Salary = 48000 },

            new Employee { ID = 6, Name = "Valarie", Gender = "Female",

                                         Department = "HR", Salary = 70000 },

            new Employee { ID = 7, Name = "John", Gender = "Male",

                                         Department = "IT", Salary = 64000 },

            new Employee { ID = 8, Name = "Pam", Gender = "Female",

                                         Department = "IT", Salary = 54000 },

            new Employee { ID = 9, Name = "Stacey", Gender = "Female",

                                         Department = "HR", Salary = 84000 },

            new Employee { ID = 10, Name = "Andy", Gender = "Male",

                                         Department = "IT", Salary = 36000 }

        };

    }

}

**Example 1:** Get Employee Count By Department

var employeeGroup = from employee in Employee.GetAllEmployees()

                    group employee by employee.Department;

foreach (var group in employeeGroup)

{

    Console.WriteLine("{0} - {1}", group.Key, group.Count());

}

**Output:**   
linq group by   
  
**Example 2:** Get Employee Count By Department and also each employee and department name

var employeeGroup = from employee in Employee.GetAllEmployees()

                                      group employee by employee.Department;

foreach (var group in employeeGroup)

{

    Console.WriteLine("{0} - {1}", group.Key, group.Count());

    Console.WriteLine("----------");

    foreach (var employee in group)

    {

        Console.WriteLine(employee.Name + "\t" + employee.Department);

    }

    Console.WriteLine(); Console.WriteLine();

}

**Output:**   
   
  
**Example 3:** Get Employee Count By Department and also each employee and department name. Data should be sorted first by Department in ascending order and then by Employee Name in ascending order.

var employeeGroup = from employee in Employee.GetAllEmployees()

                                      group employee by employee.Department into eGroup

                                      orderby eGroup.Key

                                      select new

                                      {

                                           Key = eGroup.Key,

                                           Employees = eGroup.OrderBy(x => x.Name)

                                      };

foreach (var group in employeeGroup)

{

    Console.WriteLine("{0} - {1}", group.Key, group.Employees.Count());

    Console.WriteLine("----------");

    foreach (var employee in group.Employees)

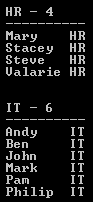
    {

        Console.WriteLine(employee.Name + "\t" + employee.Department);

    }

    Console.WriteLine(); Console.WriteLine();

}

**Output:**   


**Part 19 - Group by multiple keys in linq**

**Suggested Videos**  
[Part 16 - Cast and OfType operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-16-cast-and-oftype-operators-in.html)  
[Part 17 - AsEnumerable and AsQueryable in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-17-asenumerable-and-asqueryable-in.html)   
[Part 18 - GroupBy in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-18-groupby-in-linq.html)   
  
   
  
In this video, we will discuss **Grouping by multiple keys**. In LINQ, an anonymous type is usually used when we want to group by multiple keys.   
  
   
  
Let us understand this with an example. We will be using the following **Employee**class in this demo. This is the same class used in [Part 18](http://csharp-video-tutorials.blogspot.com/2014/07/part-18-groupby-in-linq.html). Please watch [Part 18](http://csharp-video-tutorials.blogspot.com/2014/07/part-18-groupby-in-linq.html) before proceeding.

public class Employee

{

    public int ID { get; set; }

    public string Name { get; set; }

    public string Gender { get; set; }

    public string Department { get; set; }

    public static List<Employee> GetAllEmployees()

    {

        return new List<Employee>()

        {

            new Employee { ID = 1, Name = "Mark", Gender = "Male",   
                                         Department = "IT" },

            new Employee { ID = 2, Name = "Steve", Gender = "Male",   
                                         Department = "HR" },

            new Employee { ID = 3, Name = "Ben", Gender = "Male",   
                                         Department = "IT" },

            new Employee { ID = 4, Name = "Philip", Gender = "Male",   
                                         Department = "IT" },

            new Employee { ID = 5, Name = "Mary", Gender = "Female",   
                                         Department = "HR" },

            new Employee { ID = 6, Name = "Valarie", Gender = "Female",   
                                         Department = "HR" },

            new Employee { ID = 7, Name = "John", Gender = "Male",   
                                         Department = "IT" },

            new Employee { ID = 8, Name = "Pam", Gender = "Female",   
                                         Department = "IT" },

            new Employee { ID = 9, Name = "Stacey", Gender = "Female",   
                                         Department = "HR" },

            new Employee { ID = 10, Name = "Andy", Gender = "Male",   
                                         Department = "IT" },

        };

    }

}

**Example 1:** Group employees by **Department**and then by **Gender**. The employee groups should be sorted first by **Department**and then by **Gender**in ascending order. Also, employees within each group must be sorted in ascending order by Name.

var employeeGroups = Employee.GetAllEmployees()

                                        .GroupBy(x => new { x.Department, x.Gender })

                                        .OrderBy(g => g.Key.Department).ThenBy(g => g.Key.Gender)

                                        .Select(g => new

                                        {

                                            Dept = g.Key.Department,

                                            Gender = g.Key.Gender,

                                            Employees = g.OrderBy(x => x.Name)

                                        });

foreach(var group in employeeGroups)

{

    Console.WriteLine("{0} department {1} employees count = {2}",

        group.Dept, group.Gender, group.Employees.Count());

    Console.WriteLine("--------------------------------------------");

    foreach (var employee in group.Employees)

    {

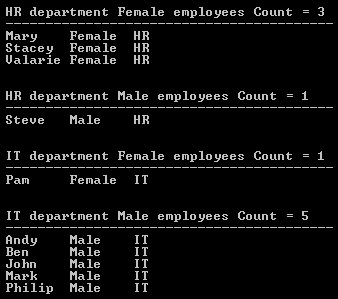
        Console.WriteLine(employee.Name + "\t" + employee.Gender

            + "\t" + employee.Department);

    }

    Console.WriteLine(); Console.WriteLine();

}

**Output:**   
   
  
**Example 2:** Rewrite Example 1 using **SQL like syntax**

varemployeeGroups=fromemployeeinEmployee.GetAllEmployees**()**

groupemployeebynew

**{**

employee.Department**,**

employee.Gender

**}**intoeGroup

orderbyeGroup.Key.Departmentascending**,**  
eGroup.Key.Genderascending

selectnew

**{**

Dept=eGroup.Key.Department**,**

Gender=eGroup.Key.Gender**,**

Employees=eGroup.OrderBy**(**x=>x.Name**)**

**};**

**Part 20 - Element Operators in LINQ**

**Suggested Videos**  
[Part 17 - AsEnumerable and AsQueryable in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-17-asenumerable-and-asqueryable-in.html)  
[Part 18 - GroupBy in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-18-groupby-in-linq.html)   
[Part 19 - Group by multiple keys in linq](http://csharp-video-tutorials.blogspot.com/2014/07/part-19-group-by-multiple-keys-in-linq.html)   
  
   
  
**The following standard query operators belong to Element Operators category**  
First / FirstOrDefault  
Last / LastOrDefault  
ElementAt / ElementAtOrDefault  
Single / SingleOrDefault  
DefaultIfEmpty   
  
   
  
**Element Operators**retrieve a single element from a sequence using the element index or based on a condition. All of these methods have a corresponding overloaded version that accepts a predicate.  
  
**First :** There are 2 overloaded versions of this method. The first overloaded version that does not have any parameters simply returns the first element of a sequence.  
  
**Example 1:**Returns the first element from the sequence

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int result = numbers.First();

Console.WriteLine("Result = " + result);

**Output:**  
Result = 1  
  
If the sequence does not contain any elements, then First() method throws an InvalidOperationException.  
  
**Example 2:**Throws InvalidOperationException.

int[] numbers = { };

int result = numbers.First();

Console.WriteLine("Result = " + result);

**Output:**  
Unhandled Exception: System.InvalidOperationException: Sequence contains no elements  
  
**The second overloaded version is used to find the first element in a sequence based on a condition.** If the sequence does not contain any elements or if no element in the sequence satisfies the condition then an InvalidOperationException is thrown.  
  
**Example 3:**Returns the first even number from the sequence

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int result = numbers.First(x => x % 2 == 0);

Console.WriteLine("Result = " + result);

**Output:**  
Result = 2  
  
**Example 4:**Throws InvalidOperationException, as no element in the sequence satisfies the condition specified by the predicate.

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int result = numbers.First(x => x % 2 == 100);

Console.WriteLine("Result = " + result);

**Output:**  
Unhandled Exception: System.InvalidOperationException: Sequence contains no matching element  
  
**FirstOrDefault :** This is very similar to First, except that this method does not throw an exception when there are no elements in the sequence or when no element satisfies the condition specified by the predicate. Instead, a default value of the type that is expected is returned. For reference types the default is NULL and for value types the default depends on the actual type expected.

**Example 5:**Returns ZERO. No element in the sequence satisfies the condition, so the default value (ZERO) for int is returned.

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int result = numbers.FirstOrDefault(x => x % 2 == 100);

Console.WriteLine("Result = " + result);

**Last :** Very similar to First, except it returns the last element of the sequence.  
  
**LastOrDefault :**Very similar to FirstOrDefault, except it returns the last element of the sequence.  
  
**ElementAt :**Returns an element at a specified index. If the sequence is empty or if the provided index value is out of range, then an ArgumentOutOfRangeException is thrown.  
  
**Example 6:**Returns element from the sequence that is at index position 1.

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

int result = numbers.ElementAt(1);

Console.WriteLine("Result = " + result);

**Output:**  
Result = 2  
  
**Example 7:**Throws ArgumentOutOfRangeException

int[] numbers = { };

int result = numbers.ElementAt(0);

Console.WriteLine("Result = " + result);

**Output:**  
Unhandled Exception: System.ArgumentOutOfRangeException: Index was out of range. Must be non-negative and less than the size of the collection.  
  
**ElementAtOrDefault :** Similar to ElementAt except that this method does not throw an exception, if the sequence is empty or if the provided index value is out of range. Instead, a default value of the type that is expected is returned.  
  
**Single :** There are 2 overloaded versions of this method. The first overloaded version that does not have any parameters returns the only element of the sequence.  
  
**Example 8:**Returns the only element (1) of the sequence.

int[] numbers = { 1 };

int result = numbers.Single();

Console.WriteLine("Result = " + result);

**Output:**  
Result = 1  
  
**Single()**method throws an exception if the sequence is empty or has more than one element.  
  
**Example 9:**Throws InvalidOperationException as the sequence contains more than ONE element.

int[] numbers = { 1, 2 };

int result = numbers.Single();

Console.WriteLine("Result = " + result);

**Output:**  
Unhandled Exception: System.InvalidOperationException: Sequence contains more than one element  
  
**The second overloaded version of the Single() method is used to find the only element in a sequence that satisfies a given condition.** An exception will be thrown if any of the following is true

**a)** If the sequence does not contain any elements OR  
**b)** If no element in the sequence satisfies the condition OR  
**c)** If more than one element in the sequence satisfies the condition  
  
**Example 10:**Throws InvalidOperationException as more than one element in the sequence satisfies the condition

int[] numbers = { 1, 2, 4 };

int result = numbers.Single(x => x % 2 == 0);

Console.WriteLine("Result = " + result);

**Output:**  
Unhandled Exception: System.InvalidOperationException: Sequence contains more than one matching element  
  
**SingleOrDefault :**Very similar to Single(), except this method does not throw an exception when the sequence is empty or when no element in the sequence satisfies the given condition. Just like Single(), this method will still throw an exception, if more than one element in the sequence satisfies the given condition.  
  
**Example 11:**Throws InvalidOperationException as more than one element in the sequence satisfies the given condition

int[] numbers = { 1, 2, 4 };

int result = numbers.SingleOrDefault(x => x % 2 == 0);

Console.WriteLine("Result = " + result);

**Output:**  
Unhandled Exception: System.InvalidOperationException: Sequence contains more than one matching element  
  
**DefaultIfEmpty :** If the sequence on which this method is called is not empty, then the values of the original sequence are returned.  
  
**Example 12 :**Returns a copy of the original sequence

int[] numbers = { 1, 2, 3 };

IEnumerable<int> result = numbers.DefaultIfEmpty();

foreach (int i in result)

{

    Console.WriteLine(i);

}

**Output:**  
1  
2  
3  
  
If the sequence is empty, then **DefaultIfEmpty**() returns a sequence with the default value of the expected type.  
  
**Example 13 :**Since the sequence is empty, a sequence containing the default value (ZERO) of int is returned.

int[] numbers = { };

IEnumerable<int> result = numbers.DefaultIfEmpty();

foreach (int i in result)

{

    Console.WriteLine(i);

}

Output:  
0  
  
The other overloaded version with a parameter allows us to specify a default value. If this method is called on a sequence that is not empty, then the values of the original sequence are returned. If the sequence is empty, then this method returns a sequence with the specified defualt value.  
  
**Example 14 :** Since the sequence is empty, a sequence containing the specified default value (10) is returned.

int[] numbers = { };

IEnumerable<int> result = numbers.DefaultIfEmpty(10);

foreach (int i in result)

{

    Console.WriteLine(i);

}

**Output:**  
10

**Part 21 - Group Join in LINQ**

**Suggested Videos**  
[Part 18 - GroupBy in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-18-groupby-in-linq.html)  
[Part 19 - Group by multiple keys in linq](http://csharp-video-tutorials.blogspot.com/2014/07/part-19-group-by-multiple-keys-in-linq.html)   
[Part 20 - Element Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-20-element-operators-in-linq.html)   
  
   
  
**The following are the different types of joins in LINQ**  
Group Join - We will discuss in this video  
Inner Join - Discussed in [Part 22](http://csharp-video-tutorials.blogspot.com/2014/08/part-22-inner-join-in-linq.html)  
Left Outer Join  
Cross Join   
  
   
  
In this video, we will discuss **Group Join**. Group Join produces hierarchical data structures. Each element from the first collection is paired with a set of correlated elements from the second collection.   
  
Let us understand **Group Join**with an **example.**Consider the following **Department**and **Employee**classes. A Department may have ZERO or MORE employees.

public class Department

{

    public int ID { get; set; }

    public string Name { get; set; }

    public static List<Department> GetAllDepartments()

    {

        return new List<Department>()

        {

            new Department { ID = 1, Name = "IT"},

            new Department { ID = 2, Name = "HR"},

            new Department { ID = 3, Name = "Payroll"},

        };

    }

}

public class Employee

{

    public int ID { get; set; }

    public string Name { get; set; }

    public int DepartmentID { get; set; }

    public static List<Employee> GetAllEmployees()

    {

        return new List<Employee>()

        {

            new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },

            new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },

            new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },

            new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },

            new Employee { ID = 5, Name = "Mary", DepartmentID = 2 },

            new Employee { ID = 6, Name = "Valarie", DepartmentID = 2 },

            new Employee { ID = 7, Name = "John", DepartmentID = 1 },

            new Employee { ID = 8, Name = "Pam", DepartmentID = 1 },

            new Employee { ID = 9, Name = "Stacey", DepartmentID = 2 },

            new Employee { ID = 10, Name = "Andy", DepartmentID = 1}

        };

    }

}

**Example 1:** Group **employees**by **Department**.

var employeesByDepartment = Department.GetAllDepartments()

                                                                           .GroupJoin(Employee.GetAllEmployees(),

                                                                             d => d.ID,

                                                                             e => e.DepartmentID,

                                                                             (department, employees) => new

                                                                             {

                                                                                 Department = department,

                                                                                 Employees = employees

                                                                             });

foreach (var department in employeesByDepartment)

{

    Console.WriteLine(department.Department.Name);

    foreach (var employee in department.Employees)

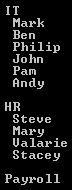
    {

        Console.WriteLine(" " + employee.Name);

    }

    Console.WriteLine();

}

**Output:**   
   
  
**Example 2:** Rewrite **Example 1**using SQL like syntax.

var employeesByDepartment = from d in Department.GetAllDepartments()

                                                       join e in Employee.GetAllEmployees()

                                                       on d.ID equals e.DepartmentID into eGroup

                                                       select new

                                                       {

                                                          Department = d,

                                                          Employees = eGroup

                                                       };

**Please note:**Group Join uses the **join**operator and the **into**keyword to group the results of the join.

**Part 22 - Inner Join in LINQ**

**Suggested Videos**  
[Part 19 - Group by multiple keys in linq](http://csharp-video-tutorials.blogspot.com/2014/07/part-19-group-by-multiple-keys-in-linq.html)  
[Part 20 - Element Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-20-element-operators-in-linq.html)   
[Part 21 - Group Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-21-group-join-in-linq.html)   
  
   
  
The following are the different types of joins in LINQ  
**Group Join** - Discussed in [Part 21](http://csharp-video-tutorials.blogspot.com/2014/07/part-21-group-join-in-linq.html)  
**Inner Join** - We will discuss in this video  
**Left Outer Join** - Later Video  
**Cross Join** - Later Video   
  
   
  
In this video we will discuss implementing **INNER JOIN**in **LINQ**. If you have 2 collections, and when you perform an inner join, then only the matching elements between the 2 collections are included in the result set. Non - Matching elements are excluded from the result set.  
  
Let us understand Inner Join with an example. Consider the following **Department**and **Employee**classes. Notice that, **Employee**Andy does not have a department assigned. An inner join will not include his record in the result set.

public class Department

{

    public int ID { get; set; }

    public string Name { get; set; }

    public static List<Department> GetAllDepartments()

    {

        return new List<Department>()

        {

            new Department { ID = 1, Name = "IT"},

            new Department { ID = 2, Name = "HR"},

            new Department { ID = 3, Name = "Payroll"},

        };

    }

}

public class Employee

{

    public int ID { get; set; }

    public string Name { get; set; }

    public int DepartmentID { get; set; }

    public static List<Employee> GetAllEmployees()

    {

        return new List<Employee>()

        {

            new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },

            new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },

            new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },

            new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },

            new Employee { ID = 5, Name = "Mary", DepartmentID = 2 },

            new Employee { ID = 6, Name = "Valarie", DepartmentID = 2 },

            new Employee { ID = 7, Name = "John", DepartmentID = 1 },

            new Employee { ID = 8, Name = "Pam", DepartmentID = 1 },

            new Employee { ID = 9, Name = "Stacey", DepartmentID = 2 },

            new Employee { ID = 10, Name = "Andy"}

        };

    }

}

**Example 1 :** Join the **Employees**and **Department**collections and print all the Employees and their respective department names.

var result = Employee.GetAllEmployees().Join(Department.GetAllDepartments(),

                                        e => e.DepartmentID,

                                        d => d.ID, (employee, department) => new

                                        {

                                            EmployeeName = employee.Name,

                                            DepartmentName = department.Name

                                        });

foreach (var employee in result)

{

    Console.WriteLine(employee.EmployeeName + "\t" + employee.DepartmentName);

}

**Output:**Notice that, in the output we don't have **Andy**record. This is because, Andy does not have a matching department in Department collection. So this is effectively an **inner join**.   
   
  
**Example 2 :**Rewrite Example 1 using SQL like syntax.

var result = from e in Employee.GetAllEmployees()

                    join d in Department.GetAllDepartments()

                    on e.DepartmentID equals d.ID

                    select new

                    {

                        EmployeeName = e.Name,

                        DepartmentName = d.Name

                    };

foreach (var employee in result)

{

    Console.WriteLine(employee.EmployeeName + "\t" + employee.DepartmentName);

}

**Part 23 - Difference between group join and inner join in linq**

**Suggested Videos**  
[Part 20 - Element Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-20-element-operators-in-linq.html)  
[Part 21 - Group Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-21-group-join-in-linq.html)   
[Part 22 - Inner Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-22-inner-join-in-linq.html)   
  
   
  
In this video, we will discuss the **difference between Group Join and Inner Join in LINQ** with examples. We will be using the following Department and Employee classes in this video. 

public class Department

{

    public int ID { get; set; }

    public string Name { get; set; }

    public static List<Department> GetAllDepartments()

    {

        return new List<Department>()

        {

            new Department { ID = 1, Name = "IT"},

            new Department { ID = 2, Name = "HR"},

            new Department { ID = 3, Name = "XX"},

        };

    }

}

public class Employee

{

    public int ID { get; set; }

    public string Name { get; set; }

    public int DepartmentID { get; set; }

    public static List<Employee> GetAllEmployees()

    {

        return new List<Employee>()

        {

            new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },

            new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },

            new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },

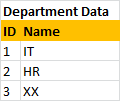
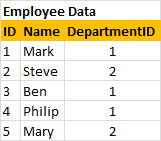
            new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },

            new Employee { ID = 5, Name = "Mary", DepartmentID = 2 }

        };

    }

}

**Department**data returned by **GetAllDepartments()**method is shown below   
   
  
**Employee**data returned by **GetAllEmployees()**method is shown below   
   
  
**The following query performs a GroupJoin on the 2 lists**

var result = from d in Department.GetAllDepartments()

                    join e in Employee.GetAllEmployees()

                    on d.ID equals e.DepartmentID into eGroup

                    select new

                    {

                       Department = d,

                       Employees = eGroup

                    };

Notice that we are using the **join**operator and the into keyword to group the results of the join. To perform group join using extension method syntax, we use **GroupJoin()**Extension method as shown below.

var result = Department.GetAllDepartments()

                                        .GroupJoin(Employee.GetAllEmployees(),

                                         d => d.ID,

                                         e => e.DepartmentID,

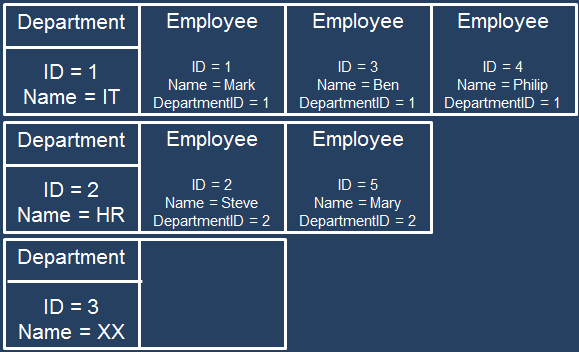
                                         (department, employees) => new

                                         {

                                              Department = department,

                                              Employees = employees

                                         });

The above 2 queries **groups employees by department**and would produce the following groups.   
  
  
To print the **Department**and **Employee**Names we use 2 foreach loops as shown below.

foreach (var department in result)

{

    Console.WriteLine(department.Department.Name);

    foreach (var employee in department.Employees)

    {

        Console.WriteLine(" " + employee.Name);

    }

    Console.WriteLine();

}

The following query performs an **Inner Join**on the 2 lists

var result = from e in Employee.GetAllEmployees()

                    join d in Department.GetAllDepartments()

                    on e.DepartmentID equals d.ID

                    select new { e, d };

To perform an **inner join**using extension method syntax, we use **Join()**Extension method as shown below.

var result = Employee.GetAllEmployees()

                                     .Join(Department.GetAllDepartments(),

                                      e => e.DepartmentID,

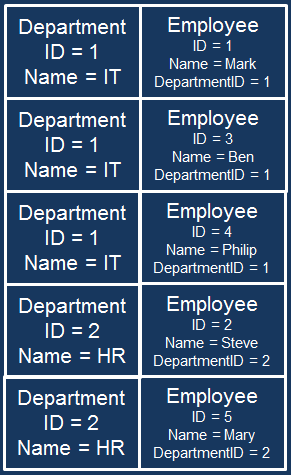
                                      d => d.ID, (employee, department) => new

                                      {

                                           e = employee,

                                           d = department

                                      });

The above 2 queries would produce a **flat result set**as shown below   
   
  
To print the **Department**and **Employee**Names we use just 1 foreach loop as shown below.

foreach (var employee in result)

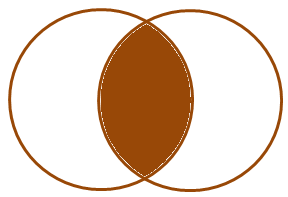
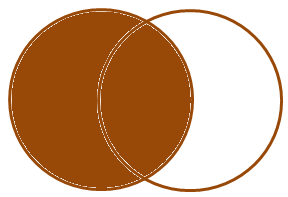
{

    Console.WriteLine(employee.e.Name + "\t" + employee.d.Name);

}

In short, **Join**is similar to **INNER JOIN**in SQL and **GroupJoin**is similar to **OUTER JOIN**in SQL

**Part 24 - Left Outer Join in LINQ**

**Suggested Videos**  
[Part 21 - Group Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/07/part-21-group-join-in-linq.html)  
[Part 22 - Inner Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-22-inner-join-in-linq.html)   
[Part 23 - Difference between group join and inner join in linq](http://csharp-video-tutorials.blogspot.com/2014/08/part-23-difference-between-group-join.html)   
  
   
  
**The following are the different types of joins in LINQ**  
Group Join - Discussed in [Part 21](http://csharp-video-tutorials.blogspot.com/2014/07/part-21-group-join-in-linq.html)  
Inner Join - Discussed in [Part 22](http://csharp-video-tutorials.blogspot.com/2014/08/part-22-inner-join-in-linq.html)  
Left Outer Join - We will discuss in this video  
Cross Join - Later Video   
  
   
  
In this video we will discuss implementing **LEFT OUTER JOIN in LINQ**.   
  
With **INNER JOIN**only the matching elements are included in the result set. Non-matching elements are excluded from the result set.   
   
  
With **LEFT OUTER JOIN**all the matching elements + all the non matching elements from the left collection are included in the result set.   
   
  
Let us understand implementing **Left Outer Join**with an example. Consider the following **Department**and **Employee**classes. Notice that, Employee Mary does not have a department assigned. An inner join will not include her record in the result set, where as a Left Outer Join will.

public class Department

{

    public int ID { get; set; }

    public string Name { get; set; }

    public static List<Department> GetAllDepartments()

    {

        return new List<Department>()

        {

            new Department { ID = 1, Name = "IT"},

            new Department { ID = 2, Name = "HR"},

        };

    }

}

public class Employee

{

    public int ID { get; set; }

    public string Name { get; set; }

    public int DepartmentID { get; set; }

    public static List<Employee> GetAllEmployees()

    {

        return new List<Employee>()

        {

            new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },

            new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },

            new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },

            new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },

            new Employee { ID = 5, Name = "Mary" }

        };

    }

}

Use **DefaultIfEmpty**() method on the results of a group join to implement **Left Outer Join**  
  
**Example 1** : Implement a **Left Outer Join**between **Employees**and **Department**collections and print all the Employees and their respective department names. Employees without a department, should display **"No Department"**against their name.

var result = from e in Employee.GetAllEmployees()

                    join d in Department.GetAllDepartments()

                    on e.DepartmentID equals d.ID into eGroup

                    from d in eGroup.DefaultIfEmpty()

                    select new

                    {

                         EmployeeName = e.Name,

                         DepartmentName = d == null ? "No Department" : d.Name

                    };

foreach (var v in result)

{

    Console.WriteLine(v.EmployeeName + "\t" + v.DepartmentName);

}

**Output:** Notice that, we also have **Mary**record in spite of she not having a department. So this is effectively a left outer join.   
   
  
**Example 2 :** Rewrite **Example 1**using extension method syntax.

var result = Employee.GetAllEmployees()

                        .GroupJoin(Department.GetAllDepartments(),

                                e => e.DepartmentID,

                                d => d.ID,

                                (emp, depts) => new { emp, depts })

                        .SelectMany(z => z.depts.DefaultIfEmpty(),

                                (a, b) => new

                                {

                                        EmployeeName = a.emp.Name,

                                        DepartmentName = b == null ? "No Department" : b.Name

                                });

foreach (var v in result)

{

    Console.WriteLine(" " + v.EmployeeName + "\t" + v.DepartmentName);

}

To implement **Left Outer Join**, with extension method syntax we use the **GroupJoin()**method along with **SelectMany()**and **DefaultIfEmpty()**methods.

**Part 25 - Cross Join in LINQ**

**Suggested Videos**  
[Part 22 - Inner Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-22-inner-join-in-linq.html)  
[Part 23 - Difference between group join and inner join in linq](http://csharp-video-tutorials.blogspot.com/2014/08/part-23-difference-between-group-join.html)   
[Part 24 - Left Outer Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-24-left-outer-join-in-linq.html)   
  
   
  
**The following are the different types of joins in LINQ**  
Group Join - [Part 21](http://csharp-video-tutorials.blogspot.com/2014/07/part-21-group-join-in-linq.html)  
Inner Join - [Part 22](http://csharp-video-tutorials.blogspot.com/2014/08/part-22-inner-join-in-linq.html)  
Left Outer Join - [Part 24](http://csharp-video-tutorials.blogspot.com/2014/08/part-24-left-outer-join-in-linq.html)  
Cross Join - We will discuss in this video   
  
   
  
In this video we will discuss implementing **CROSS JOIN in LINQ**.   
  
**Cross join** produces a cartesian product i.e when we cross join two sequences, every element in the first collection is combined with every element in the second collection. The total number of elements in the resultant sequence will always be equal to the product of the elements in the two source sequences. The on keyword that specfies the JOIN KEY is not required.  
  
Let us understand implementing Cross Join with an example. Consider the following **Department**and **Employee**classes. 

public class Department

{

    public int ID { get; set; }

    public string Name { get; set; }

    public static List<Department> GetAllDepartments()

    {

        return new List<Department>()

        {

            new Department { ID = 1, Name = "IT"},

            new Department { ID = 2, Name = "HR"},

        };

    }

}

public class Employee

{

    public int ID { get; set; }

    public string Name { get; set; }

    public int DepartmentID { get; set; }

    public static List<Employee> GetAllEmployees()

    {

        return new List<Employee>()

        {

            new Employee { ID = 1, Name = "Mark", DepartmentID = 1 },

            new Employee { ID = 2, Name = "Steve", DepartmentID = 2 },

            new Employee { ID = 3, Name = "Ben", DepartmentID = 1 },

            new Employee { ID = 4, Name = "Philip", DepartmentID = 1 },

            new Employee { ID = 5, Name = "Mary", DepartmentID = 2 },

        };

    }

}

**Example 1 :**Cross Join **Employees**collection with **Departments**collections.

var result = from e in Employee.GetAllEmployees()

                    from d in Department.GetAllDepartments()

                    select new { e, d };

foreach (var v in result)

{

    Console.WriteLine(v.e.Name + "\t" + v.d.Name);

}

**Output:**We have 5 elements in **Employees**collection and 2 elements in **Departments**collection. In the result we have 10 elements, i.e the cartesian product of the elements present in Employees and Departments collection. Notice that every element from the Employees collection is combined with every element in the Departments collection.   
   
  
**Example 2 :** Cross Join **Departments**collections with **Employees**collection

var result = from d in Department.GetAllDepartments()

                    from e in Employee.GetAllEmployees()

                    select new { e, d };

foreach (var v in result)

{

    Console.WriteLine(v.e.Name + "\t" + v.d.Name);

}

**Output:** Notice that the output in this case is slightly different from **Example 1**. In this case, every element from the Departments collection is combined with every element in the Employees collection.   
   
  
**Example 3 :** Rewrite **Example 1**using extension method syntax  
  
To implement **Cross Join**using extension method syntax, we could either use SelectMany() method or Join() method  
  
**Implementing cross join using SelectMany()**

var result = Employee.GetAllEmployees()

                        .SelectMany(e => Department.GetAllDepartments(), (e, d) => new { e, d });

foreach (var v in result)

{

    Console.WriteLine(v.e.Name + "\t" + v.d.Name);

}

**Implementing cross join using Join()**

var result = Employee.GetAllEmployees()

                                     .Join(Department.GetAllDepartments(),

                                               e => true,

                                               d => true,

                                               (e, d) => new { e, d });

foreach (var v in result)

{

    Console.WriteLine(v.e.Name + "\t" + v.d.Name);

}

**Part 26 - Set operators in LINQ**

**Suggested Videos**  
[Part 23 - Difference between group join and inner join in linq](http://csharp-video-tutorials.blogspot.com/2014/08/part-23-difference-between-group-join.html)  
[Part 24 - Left Outer Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-24-left-outer-join-in-linq.html)   
[Part 25 - Cross Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-25-cross-join-in-linq.html)   
  
   
  
**The following operators belong to Set operators category**  
Distinct  
Union  
Intersect  
Except   
  
   
  
In this video we will discuss **Distinct**operator. This operator returns distinct elements from a given collection.  
  
**Example 1:**Return **distinct**country names. In this example the default comparer is being used and the comparison is case-sensitive, so in the output we see country USA 2 times.

string[] countries = { "USA", "usa", "INDIA", "UK", "UK" };

var result = countries.Distinct();

foreach (var v in result)

{

    Console.WriteLine(v);

}

**Output:**   
distinct in linq   
  
**Example 2:** For the **comparison to be case-insensitive**, use the other overloaded version of **Distinct()**method to which we can pass a class that implements **IEqualityComparer**as an argument. In this case we see country USA only once in the output.

string[] countries = { "USA", "usa", "INDIA", "UK", "UK" };

var result = countries.Distinct(StringComparer.OrdinalIgnoreCase);

foreach (var v in result)

{

    Console.WriteLine(v);

}

**Output:**   
distinct in linq example   
  
When comparing elements, **Distinct**() works in a slightly different manner with **complex types**like Employee, Customer etc.   
  
**Example 3:**Notice that in the output we don't get unique employees. This is because, the default comparer is being used which will just check for object references being equal and not the individual property values.

List<Employee> list = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 102, Name = "Mary"}

};

var result = list.Distinct();

foreach (var v in result)

{

    Console.WriteLine(v.ID + "\t" + v.Name);

}

**Output:**   
linq distinct c# example   
  
**To solve the problem in Example 3, there are 3 ways**  
**1.** Use the other overloaded version of **Distinct()** method to which we can pass a custom class that implements **IEqualityComparer**

**2.** Override **Equals()**and **GetHashCode()**methods in **Employee**class  
**3.** Project the properties into a **new anonymous type**, which overrides **Equals()**and **GetHashCode()**methods  
  
**Example 4 :**Using the overloaded version of **Distinct()**method to which we can pass a custom class that implements **IEqualityComparer**  
  
**Step 1 :**Create a custom class that implements **IEqualityComparer<T>** and implement **Equals()**and **GetHashCode()**methods

public class EmployeeComparer : IEqualityComparer<Employee>

{

    public bool Equals(Employee x, Employee y)

    {

        return x.ID == y.ID && x.Name == y.Name;

    }

    public int GetHashCode(Employee obj)

    {

        return obj.ID.GetHashCode() ^ obj.Name.GetHashCode();

    }

}

**Step 2 :** Pass an instance of **EmployeeComparer**as an argument to **Distinct()**method

List<Employee> list = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 102, Name = "Mary"}

};

var result = list.Distinct(new EmployeeComparer());

foreach (var v in result)

{

    Console.WriteLine(v.ID + "\t" + v.Name);

}

**Output:**   
iequalitycomparer example   
  
**Example 5 :**Override **Equals()**and **GetHashCode()**methods in **Employee**class

public class Employee

{

    public int ID { get; set; }

    public string Name { get; set; }

    public override bool Equals(object obj)

    {

        return this.ID == ((Employee)obj).ID && this.Name == ((Employee)obj).Name;

    }

    public override int GetHashCode()

    {

        return this.ID.GetHashCode() ^ this.Name.GetHashCode();

    }

}

**Example 6 :** Project the properties into a **new anonymous type**, which overrides **Equals()**and **GetHashCode()**methods

List<Employee> list = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 102, Name = "Mary"}

};

var result = list.Select(x => new { x.ID, x.Name }).Distinct();

foreach (var v in result)

{

    Console.WriteLine(" " + v.ID + "\t" + v.Name);

}

**Part 27 - Union, Intersect and Except operators in LINQ**

**Suggested Videos**  
[Part 24 - Left Outer Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-24-left-outer-join-in-linq.html)  
[Part 25 - Cross Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-25-cross-join-in-linq.html)   
[Part 26 - Set operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-26-set-operators-in-linq.html)   
  
   
  
**The following operators belong to Set operators category**  
Distinct  
Union  
Intersect  
Except   
  
   
  
We discussed **Distinct**operator in [Part 26](http://csharp-video-tutorials.blogspot.com/2014/08/part-26-set-operators-in-linq.html). In this video we will discuss **Union**, **Intersect** and **Except**operators.  
  
**Union**combines two collections into one collection while removing the duplicate elements.  
  
**Example 1:**numbers1 and numbers2 collections are combined into a single collection. Notice that, the duplicate elements are removed.

int[] numbers1 = { 1, 2, 3, 4, 5 };

int[] numbers2 = { 1, 3, 6, 7, 8 };

var result = numbers1.Union(numbers2);

foreach (var v in result)

{

    Console.WriteLine(v);

}

**Output:**   
union in linq   
  
When **comparing elements**, just like **Distinct()**method, **Union(), Intersect()**and **Except()**methods work in a slightly different manner with **complex types**like **Employee, Customer**etc.   
  
**Example 2 :**Notice that in the output the duplicate employee objects are not removed. This is because, the default comparer is being used which will **just check for object references being equal**and not the individual property values.

List<Employee> list1 = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 102, Name = "Susy"},

    new Employee { ID = 103, Name = "Mary"}

};

List<Employee> list2 = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 104, Name = "John"}

};

var result = list1.Union(list2);

foreach (var v in result)

{

    Console.WriteLine(v.ID + "\t" + v.Name);

}

**Output :**   
   
  
**Example 3 :**To solve the problem in **Example 2**, there are 3 ways  
**1.** Use the other overloaded version of **Union()**method to which we can pass a custom class that implements **IEqualityComparer**

**2.**Override **Equals()**and **GetHashCode()**methods in **Employee**class  
**3.** Project the properties into a new anonymous type, which overrides **Equals()**and **GetHashCode()**methods  
  
Project the properties into a new anonymous type, which overrides **Equals()**and **GetHashCode()**methods

List<Employee> list1 = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 102, Name = "Susy"},

    new Employee { ID = 103, Name = "Mary"}

};

List<Employee> list2 = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 104, Name = "John"}

};

var result = list1.Select(x => new { x.ID, x.Name })

                    .Union(list2.Select(x => new { x.ID, x.Name }));

foreach (var v in result)

{

    Console.WriteLine(v.ID + "\t" + v.Name);

}

**Output :**   
union in linq c#   
  
**Intersect()**returns the common elements between the 2 collections.  
  
**Example 4 :**Return common elements in numbers1 and numbers2 collections.

int[] numbers1 = { 1, 2, 3, 4, 5 };

int[] numbers2 = { 1, 3, 6, 7, 8 };

var result = numbers1.Intersect(numbers2);

foreach (var v in result)

{

    Console.WriteLine(v);

}

**Output :**   
intersect in linq   
  
**Except()** returns the elements that are present in the first collection but not in the second collection.  
  
**Example 5:**Return the elements that are present in the first collection but not in the second collection.

int[] numbers1 = { 1, 2, 3, 4, 5 };

int[] numbers2 = { 1, 3, 6, 7, 8 };

var result = numbers1.Except(numbers2);

foreach (var v in result)

{

    Console.WriteLine(v);

}

**Output :**   
except in linq

**Part 28 - Generation Operators in LINQ**

**Suggested Videos**  
[Part 25 - Cross Join in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-25-cross-join-in-linq.html)  
[Part 26 - Set operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-26-set-operators-in-linq.html)   
[Part 27 - Union, Intersect and Except operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-27-union-intersect-and-except.html)   
  
   
  
The following operators belong to **Generation Operators**category  
Range  
Repeat  
Empty   
  
   
  
**Range operator generates a sequence of integers within a specified range.** This method has 2 integer parameters. The start parameter specifies the integer to start with and the count parameter specifies the number of sequential integers to generate.  
  
For example to print the first 10 even numbers without using LINQ, we would use a for loop as shown below.

for (int i = 1; i <= 10; i++)

{

    if (i % 2 == 0)

    {

        Console.WriteLine(i);

    }

}

To achieve the same using LINQ, we can use **Range**method as shown below.

var evenNumbers = Enumerable.Range(1, 10).Where(x => x % 2 == 0);

foreach (int i in evenNumbers)

{

    Console.WriteLine(i);

}

Output :    
linq range example   
  
**Repeat**operator is used to generate a sequence that contains one repeated value.  
  
**For example**the following code returns a string sequence that contains **5 "Hello" string objects**in it.

var result = Enumerable.Repeat("Hello", 5);

foreach (var v in result)

{

    Console.WriteLine(v);

}

**Output:**   
linq repeat n times   
  
**Empty operator returns an empty sequence of the specified type**. For example  
Enumerable.Empty<int>() - Returns an empty IEnumerable<int>  
Enumerable.Empty<string>() - Returns an empty IEnumerable<string>  
  
The question that comes to our mind is, **what is the use of Empty() method**. Here is an example where we could use Empty() method  
  
There may be scenarios where our application calls a method in a third party application that returns **IEnumerable<int>**. There may be a situation where the third party method returns null. For the purpose of this example, let us assume the third party method is similar to **GetIntegerSequence().**  
  
A NULL reference exception will be thrown if we run the following code

class Program

{

    public static void Main()

    {

        IEnumerable<int> result = GetIntegerSequence();

        foreach (var v in result)

        {

            Console.WriteLine(v);

        }

    }

    private static IEnumerable<int> GetIntegerSequence()

    {

        return null;

    }

}

One way to fix this is to **check for NULL**before looping thru the items in the result as shown below.

class Program

{

    public static void Main()

    {

        IEnumerable<int> result = GetIntegerSequence();

        if (result != null)

        {

            foreach (var v in result)

            {

                Console.WriteLine(v);

            }

        }

    }

    private static IEnumerable<int> GetIntegerSequence()

    {

        return null;

    }

}

The other way to fix it, is by using **Empty()**linq method as shown below. Here we are using **NULL-COALESCING operator**that checks if the **GetIntegerSequence()**method returns NULL, in which case the result variable is initialized with an empty **IEnumerable<int>.**

class Program

{

    public static void Main()

    {

        IEnumerable<int> result = GetIntegerSequence() ?? Enumerable.Empty<int>();

        foreach (var v in result)

        {

            Console.WriteLine(v);

        }

    }

    private static IEnumerable<int> GetIntegerSequence()

    {

        return null;

    }

}

**Part 29 - Concat operator in LINQ**

**Suggested Videos**  
[Part 26 - Set operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-26-set-operators-in-linq.html)  
[Part 27 - Union, Intersect and Except operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-27-union-intersect-and-except.html)   
[Part 28 - Generation Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-28-generation-operators-in-linq.html)   
  
   
  
**In this video we will discuss**  
**1.** The use of Concat operator  
**2.** Difference between Concat and Union operators   
  
   
  
**Concat operator concatenates two sequences into one sequence.**  
  
The following code will **concatenate both the integer sequences** (numbers1 & numbers2) into one integer sequence. Notice that the duplicate elements ARE NOT REMOVED.

int[] numbers1 = { 1, 2, 3 };

int[] numbers2 = { 1, 4, 5 };

var result = numbers1.Concat(numbers2);

foreach (var v in result)

{

    Console.WriteLine(v);

}

**Output :**   
linq concat example   
  
Now let us perform a **union**between the 2 integer sequences (numbers1 & numbers2). Just like concat operator, union operator also combines the 2 integer sequences (numbers1 & numbers2) into one integer sequence, but notice that the **duplicate elements ARE REMOVED.**

int[] numbers1 = { 1, 2, 3 };

int[] numbers2 = { 1, 4, 5 };

var result = numbers1.Union(numbers2);

foreach (var v in result)

{

    Console.WriteLine(v);

}

**Output :**   
linq concat example c#   
  
**What is the difference between Concat and Union operators?**  
Concat operator combines 2 sequences into 1 sequence. Duplicate elements are not removed. It simply returns the items from the first sequence followed by the items from the second sequence.   
  
Union operator also combines 2 sequences into 1 sequence, but will remove the duplicate elements.

**Part 30 - SequenceEqual Operator in LINQ**

**Suggested Videos**  
[Part 27 - Union, Intersect and Except operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-27-union-intersect-and-except.html)  
[Part 28 - Generation Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-28-generation-operators-in-linq.html)   
[Part 29 - Concat operator in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-29-concat-operator-in-linq.html)   
  
   
  
**SequenceEqual() method is used to determine whether two sequences are equal.**This method returns true if the sequences are equal otherwise false.    
  
   
  
**For 2 sequences to be equal**  
**1.** Both the sequences should have same number of elements and  
**2.** Same values should be present in the same order in both the sequences  
  
**Example 1 :**SequenceEqual() returns true.  
  
string[] countries1 = { "USA", "India", "UK" };

string[] countries2 = { "USA", "India", "UK" };

var result = countries1.SequenceEqual(countries2);

Console.WriteLine("Are Equal = " + result);

**Example 2 :** In this case, **SequenceEqual()**returns false, as the default comparison is case sensitive.   
  
string[] countries1 = { "USA", "INDIA", "UK" };

string[] countries2 = { "usa", "india", "uk" };

var result = countries1.SequenceEqual(countries2);

Console.WriteLine("Are Equal = " + result);

**Example 3:**If we want the comparison to be **case-insensitive**, then use the other overloaded version of SequenceEqual() method to which we can pass an alternate comparer.  
  
string[] countries1 = { "USA", "INDIA", "UK" };

string[] countries2 = { "usa", "india", "uk" };

var result = countries1.SequenceEqual(countries2,StringComparer.OrdinalIgnoreCase);

Console.WriteLine("Are Equal = " + result);

**Example 4 :** SequenceEqual() returns false. This is because, although both the sequences contain same data, the data is not present in the same order.  
  
string[] countries1 = { "USA", "INDIA", "UK" };

string[] countries2 = { "UK", "INDIA", "USA" };

var result = countries1.SequenceEqual(countries2);

Console.WriteLine("Are Equal = " + result);

**Example 5 :** To fix the problem in Example 4, use **OrderBy()**to sort data in the source sequences.  
  
string[] countries1 = { "USA", "INDIA", "UK" };

string[] countries2 = { "UK", "INDIA", "USA" };

var result = countries1.OrderBy(c => c).SequenceEqual(countries2.OrderBy(c => c));

Console.WriteLine("Are Equal = " + result);

**Example 6 :** When comparing complex types, the default comparer will only check if the object references are equal. So, in this case SequenceEqual() returns false.  
  
List<Employee> list1 = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 102, Name = "Susy"},

};

List<Employee> list2 = new List<Employee>()

{

    new Employee { ID = 101, Name = "Mike"},

    new Employee { ID = 102, Name = "Susy"},

};

var result = list1.SequenceEqual(list2);

Console.WriteLine("Are Equal = " + result);

**To solve the problem in Example 6, there are 3 ways**  
**1.** Use the other overloaded version of SequenceEqual() method to which we can pass a custom class that implements IEqualityComparer

**2.** Override Equals() and GetHashCode() methods in Employee class  
**3.** Project the properties into a new anonymous type, which overrides Equals() and GetHashCode() methods  
  
We discussed implementing these 3 options for Distinct() method in [Part 26](http://csharp-video-tutorials.blogspot.com/2014/08/part-26-set-operators-in-linq.html) of [LINQ Tutorial](https://www.youtube.com/playlist?list=PL6n9fhu94yhWi8K02Eqxp3Xyh_OmQ0Rp6). In the same way these options can be implemented for SequenceEqual() method.

**Part 31 - Quantifiers in LINQ**

**Suggested Videos**  
[Part 28 - Generation Operators in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-28-generation-operators-in-linq.html)  
[Part 29 - Concat operator in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-29-concat-operator-in-linq.html)   
[Part 30 - SequenceEqual Operator in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-30-sequenceequal-operator-in-linq.html)   
  
   
  
**The following methods belong to Quantifiers category**  
All  
Any  
Contains   
  
   
  
All these methods return true or false depending on whether if some or all of the elements in a sequence satisfy a condition.  
  
**All()**method returns true if all the elements in a sequence satisfy a given condition, otherwise false.  
  
**Example 1 :**Returns true, as all the numbers are less than 10

int[] numbers = { 1, 2, 3, 4, 5 };

var result = numbers.All(x => x < 10);

Console.WriteLine(result);

There are 2 overloaded versions of **Any()**method. The version without any parameters checks if the sequence contains at least one element. The other version with a predicate parameter checks if the sequence contains at least one element that satisfies a given condition.  
  
**Example 2 :**Returns true as the sequence contains at least one element

int[] numbers = { 1, 2, 3, 4, 5 };

var result = numbers.Any();

Console.WriteLine(result);

**Example 3 :** Returns false as the sequence does not contain any element that satisfies the given condition (No element in the sequence is greater than 10)

int[] numbers = { 1, 2, 3, 4, 5 };

var result = numbers.Any(x => x > 10);

Console.WriteLine(result);

There are 2 overloaded versions of the **Contains()**method. One of the overloaded version checks if the sequence contains a specified element using the default equality comparer. The other overloaded version checks if the sequence contains a specified element using an alternate equality comparer.  
  
**Example 4 :**Returns true as the sequence contains number 3. In this case the default equality comparer is used.

int[] numbers = { 1, 2, 3, 4, 5 };

var result = numbers.Contains(3);

Console.WriteLine(result);

**Example 5 :** Returns true. In this case we are using an alternate equality comparer (StringComparer) for the comparison to be case-insensitive.

string[] countries = { "USA", "INDIA", "UK" };

var result = countries.Contains("india", StringComparer.OrdinalIgnoreCase);

Console.WriteLine(result);

When comparing complex types like **Employee, Customer**etc, the default comparer will only check if the object references are equal, and not the individual property values of the objects that are being compared.  
  
**Example 6 :**Returns false, as the default comparer will only check if the object references are equal.

List<Employee> employees = new List<Employee>()

{

    new Employee { ID = 101, Name = "Rosy"},

    new Employee { ID = 102, Name = "Susy"}

};

var result = employees.Contains(new Employee { ID = 101, Name = "Rosy" });

Console.WriteLine(result);

**To solve the problem in Example 6, there are 3 ways**  
**1.** Use the other overloaded version of **Contains()**method to which we can pass a custom class that implements **IEqualityComparer**

**2.** Override **Equals()**and **GetHashCode()**methods in **Employee**class  
**3.** Project the properties into a new anonymous type, which overrides **Equals()**and **GetHashCode()**methods  
  
We discussed implementing these **3 options for Distinct() method in**[**Part 26**](http://csharp-video-tutorials.blogspot.com/2014/08/part-26-set-operators-in-linq.html)**of**[**LINQ Tutorial**](https://www.youtube.com/playlist?list=PL6n9fhu94yhWi8K02Eqxp3Xyh_OmQ0Rp6). In the same way these options can be implemented for Contains() method.

**Part 32 - LinqPad Tutorial**

**Suggested Videos**  
[Part 29 - Concat operator in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-29-concat-operator-in-linq.html)  
[Part 30 - SequenceEqual Operator in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-30-sequenceequal-operator-in-linq.html)   
[Part 31 - Quantifiers in LINQ](http://csharp-video-tutorials.blogspot.com/2014/08/part-31-quantifiers-in-linq.html)   
  
   
  
**What is LinqPad**  
**LinqPad**is a free tool that you can download from [http://www.linqpad.net](http://www.linqpad.net/). It helps learn, write and test linq queries.   
  
   
  
Copy and paste the following LINQ query in LinqPad. To execute the query, you can either press the **Green Execute button**on the LinqPad or press **F5**. **Dump()**method is similar to **Console.WriteLine()**in a console application.

int[] numbers = { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

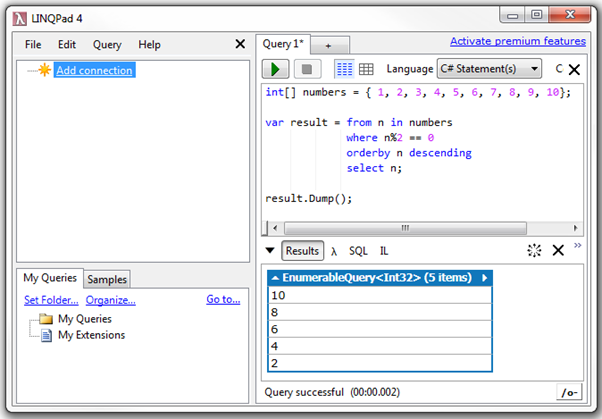
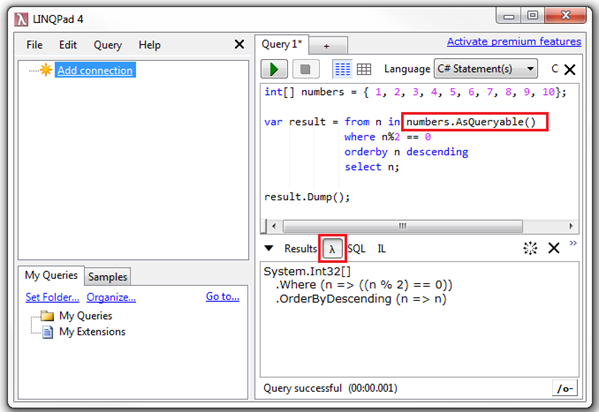
var result = from n in numbers

                where n % 2 == 0

                orderby n descending

                select n;

result.Dump();

  
  
Notice that the results of the query are shown in the **Results**window. Next to the results window, you also have the following options  
**1. ? (lambda Symbol) -**Use this button to get the lambda equivalent of a LINQ Query  
**2. SQL -**Shows the generated SQL statement that will be executed against the underlying database  
**3. IL -**Shows the Intermediate Language code  
  
For the above query, Lambda and SQL windows will not show anything. To get the Lambda equivalent of a LINQ query, use **.AsQueryable()**on the source collection as shown below.   
  
  
**AsQueryable()** can also be used on the source collection as shown below.

var numbers = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 }.AsQueryable();

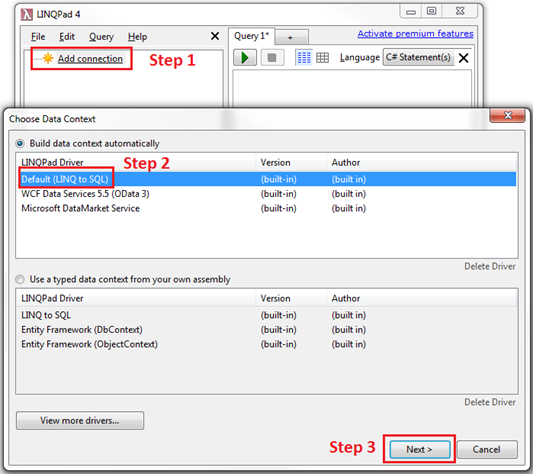
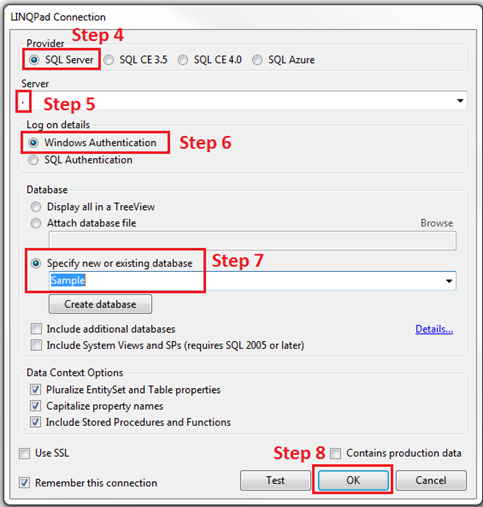
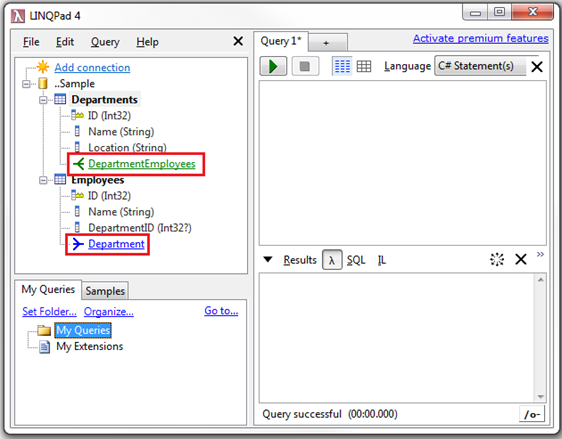
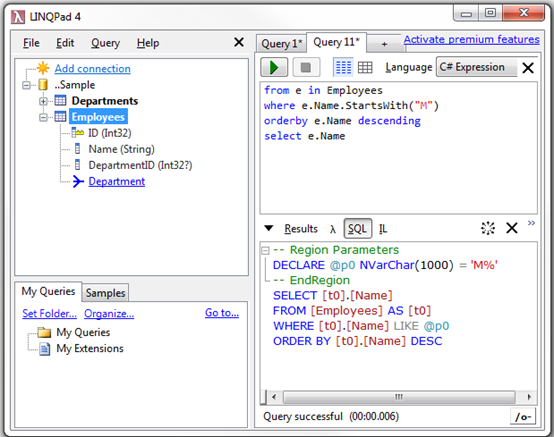
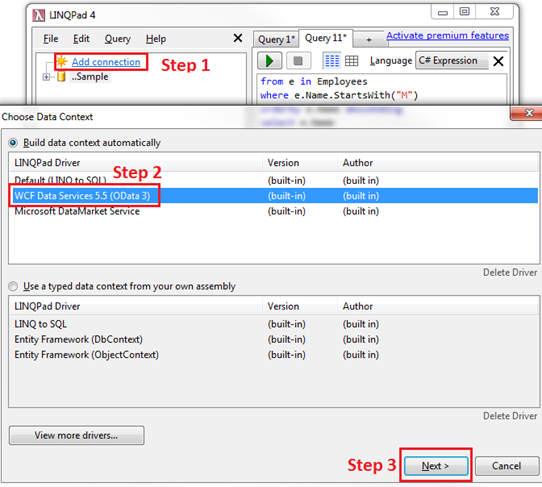
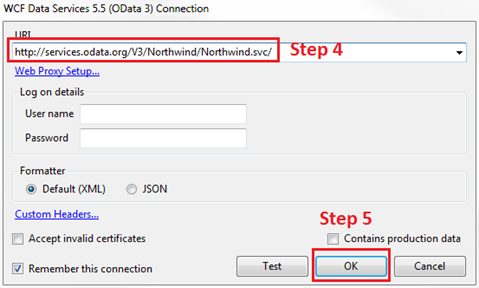
var result = from n in numbers

                where n % 2 == 0

                orderby n descending

                select n;

result.Dump();

**LinqPad can execute**  
1. Statements  
2. Expressions  
3. Program  
  
LinqPad can also be used with databases and WCF Data Services.  
  
**Adding a database connection in LinqPad**  
**Step 1 :**Click "Add connection"  
**Step 2 :**Under LinqPad Driver, select "Default (LINQ to SQL)"  
**Step 3 :**Click Next  
  
   
  
**Step 4 :** Select "SQL Server" as the "Provider"  
**Step 5 :**Specify the Server Name. In my case I am connecting to the local SQL Server. So I used . (DOT)  
**Step 6 :**Select the Authentication   
**Step 7 :**Select the Database  
**Step 8 :**Click OK   
  
   
  
At this point LinqPad connects to the database, and shows all the table entities. The relationships between the entities are also shown. The Green Split arrow indicates One-to-Many relationship and the Blue Split Arrow indicates Many-to-One relationship.    
  
   
  
We can now start writing linq queries targeting the SQL Server database.  
  
The following LINQ query fetches **all the employee names that start with letter 'M' and sorts them in descending order**  
from e in Employees  
where e.Name.StartsWith("M")  
orderby e.Name descending  
select e.Name  
  
After executing the query, click on the SQL button to see the Transact-SQL that is generated.  
   
  
**Adding a WCF Data Services connection in LinqPad**  
**Step 1 :**Click "Add connection"  
**Step 2 :**Under LinqPad Driver, select "WCF Data Services"  
**Step 3 :**Click Next   
  
   
  
**Step 4 :**Type the URI for the WCF Data Service. http://services.odata.org/V3/Northwind/Northwind.svc/  
**Step 5 :**Click OK   
  
   
  
We can now start writing linq queries targeting the WCF Data Service.  
  
The following LINQ query fetches **all the product names that start with letter 'C' and sorts them in ascending order**  
from p in Products  
where p.ProductName.StartsWith("C")  
orderby p.ProductName ascending  
select p

**Part 1 - LINQ to SQL**

In this video we will discuss **using LINQ to SQL to retrieve data from a SQL Server database**.   
  
   
  
**What is LINQ to SQL**  
**LINQ to SQL is an ORM (Object Relational Mapping) framework,** that automatically creates strongly typed .net classes based on database tables. We can then write LINQ to SQL queries (Select, Insert, Update, Delete) in any .NET supported language (C#, VB etc). The LINQ to SQL provider will then convert LINQ queries to Transact-SQL that the SQL Server database understands. LINQ to SQL supports transactions, views, and stored procedures. LINQ to SQL supports only SQL Server database.   
  
   
  
**Since LINQ to SQL models a relational database using strongly typed .net classes, we have the following advantages**  
**1.** Intellisense support  
**2.** Compile time error checking  
**3.** Debugging support  
  
**Modeling Databases - Creating LINQ to SQL classes**  
Use the LINQ to SQL designer that ships with Visual Studio to create LINQ to SQL classes. Here are the steps.  
  
**Step 1 :**Create a dataabse. Name it Sample.  
  
**Step 2 :**Execute the following SQL script to create Departments and Employees tables and populate them with test data.

Create table Departments

(

     ID int primary key identity,

     Name nvarchar(50),

     Location nvarchar(50)

)

GO

Create table Employees

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50),

     Salary int,

     DepartmentId int foreign key references Departments(Id)

)

GO

Insert into Departments values ('IT', 'New York')

Insert into Departments values ('HR', 'London')

Insert into Departments values ('Payroll', 'Sydney')

GO

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000, 1)

Insert into Employees values ('Steve', 'Pound', 'Male', 45000, 3)

Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000, 1)

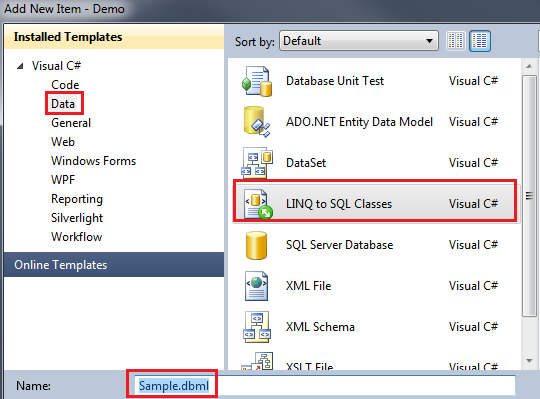
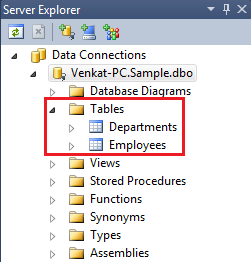
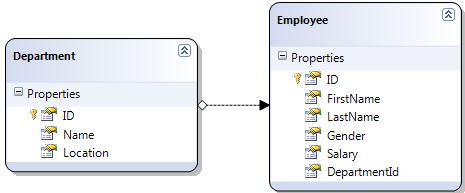
Insert into Employees values ('Philip', 'Hastings', 'Male', 45000, 2)

Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000, 2)

Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000, 3)

Insert into Employees values ('John', 'Stanmore', 'Male', 80000, 1)

GO

**Step 3 :** Run Visual Studio as an administrator. Create a new empty asp.net web application project. Name it **Demo**.  
  
**Step 4 :**Right click on the project in solution explorer and add **LINQ to SQL Classes**. Change the name from **DataClasses1.dbml**to **Sample.dbml**   
   
  
**Step 5 :** At this point, **Sample.dbml**file should have been added to the project. Click on **Server Explorer**link on **Sample.dbml**file. In the **Server Explorer**window, you should find all the tables in the **Sample**database. Drag and drop the tables on **Sample.dbml**file.   
   
  
**Step 6 :** At this point we should have **Department**and **Employee**classes. The properties of the class map to the **columns**of the respective **table**in the database. The arrow between the classes represent the **association**between them. These associations are modeled based on the **primary-key/foreign-key relationships**between the tables in the database. Notice that the arrow is pointing from **Department**to **Employee**entity. In this case there is a **One-to-Many relationship**between **Department**and **Employee**entities. A Department can have 1 or more employees.   
   
  
**Step 7 :** Add a WebForm to the project. Drag and Drop a GridView control on the webform.  
  
**Step 8 :**Copy and paste the following code in the code-behind file.

using System;

using System.Linq;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        {

            SampleDataContext dbContext = new SampleDataContext();

            GridView1.DataSource = from employee in dbContext.Employees

                                                         where employee.Gender == "Male"

                                                         orderby employee.Salary descending

                                                         select employee;

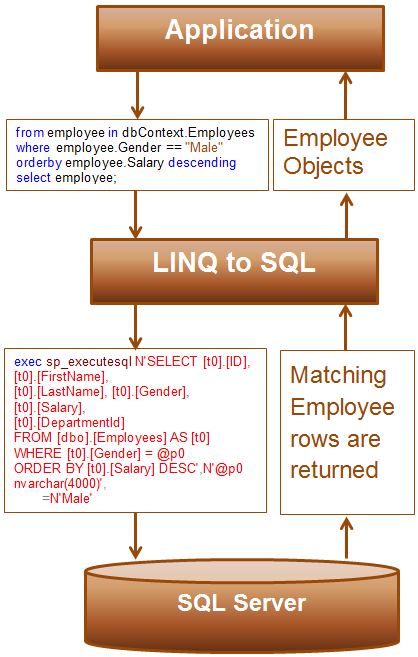
            GridView1.DataBind();

        }

    }

}

In the **Page\_Load()** event, we are creating an instance of **SampleDataContext**class. We will discuss DataContext class in detail in a later video session. For now, understand that the **DataContext**is the **entry point**to the **underlying database.**Next we have a LINQ query which fetches all the Male Employees sorted by Salary in descending order.  
  
We have not written any T-SQL code here. So, **how is the application able to retrieve data from the SQL Server database.** Let us understand what is happening behind the scenes.  
**1.** Application issues a LINQ Query  
**2.** LINQ to SQL provider translates the LINQ query into T-SQL that the SQL Server database can understand  
**3.** SQL Server executes the query, and returns the matching rows

**4.** LINQ to SQL provider creates Employee objects, populates properties and return the objects to the application.   
  


**Part 2 - Insert Update Delete using LINQ to SQL**

**Suggested Videos**  
[Part 1 - LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-1-linq-to-sql.html)   
  
   
  
In this video, we will discuss **performing Insert Update and Delete using LINQ to SQL**. We will continue with the example, we worked with in [Part 1](http://csharp-video-tutorials.blogspot.com/2014/09/part-1-linq-to-sql.html). In [Part 1](http://csharp-video-tutorials.blogspot.com/2014/09/part-1-linq-to-sql.html), we discussed performing a Select using LINQ to SQL. Please watch [Part 1](http://csharp-video-tutorials.blogspot.com/2014/09/part-1-linq-to-sql.html) before proceeding.   
  
   
  
**Insert using LINQ to SQL**

using (SampleDataContext dbContext = new SampleDataContext())

{

    Employee newEmployee = new Employee

    {

        FirstName = "Tim",

        LastName = "T",

        Gender = "Male",

        Salary = 55000,

        DepartmentId = 1

    };

    dbContext.Employees.InsertOnSubmit(newEmployee);

    dbContext.SubmitChanges();

}

**Update using LINQ to SQL**

using (SampleDataContext dbContext = new SampleDataContext())

{

    Employee employee = dbContext.Employees.SingleOrDefault(x => x.ID == 8);

    employee.Salary = 65000;

    dbContext.SubmitChanges();

}  
  
**Delete using LINQ to SQL**

using (SampleDataContext dbContext = new SampleDataContext())

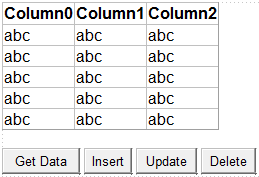
{

    Employee employee = dbContext.Employees.SingleOrDefault(x => x.ID == 8);

    dbContext.Employees.DeleteOnSubmit(employee);

    dbContext.SubmitChanges();

}

Now let us continue with the example that we worked with in [Part 1](http://csharp-video-tutorials.blogspot.com/2014/09/part-1-linq-to-sql.html).  
  
**Step 1 :**Drag and drop 4 button controls on the webform. Change the **Text**& **ID**properties of the 4 button controls as shown below.  
Button 1 : Text = Get Data, ID = btnGetData  
Button 2 : Text = Insert, ID = btnInsert  
Button 3 : Text = Update, ID = btnUpdate  
Button 4 : Text = Delete, ID = btnDelete  
  
Double click on each of the button controls to generate their respective click event handler methods.  
  
At this point, the design of the WebForm should be as shown below.   
   
  
**Step 2 :** Copy and paste the following code in the code-behind file.

using System;

using System.Linq;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        {}

        private void GetData()

        {

            SampleDataContext dbContext = new SampleDataContext();

            GridView1.DataSource = dbContext.Employees;

            GridView1.DataBind();

        }

        protected void btnGetData\_Click(object sender, EventArgs e)

        {

            GetData();

        }

        protected void btnInsert\_Click(object sender, EventArgs e)

        {

            using (SampleDataContext dbContext = new SampleDataContext())

            {

                Employee newEmployee = new Employee

                {

                    FirstName = "Tim",

                    LastName = "T",

                    Gender = "Male",

                    Salary = 55000,

                    DepartmentId = 1

                };

                dbContext.Employees.InsertOnSubmit(newEmployee);

                dbContext.SubmitChanges();

            }

            GetData();

        }

        protected void btnUpdate\_Click(object sender, EventArgs e)

        {

            using (SampleDataContext dbContext = new SampleDataContext())

            {

                Employee employee = dbContext.Employees.SingleOrDefault(x => x.ID == 8);

                employee.Salary = 65000;

                dbContext.SubmitChanges();

            }

            GetData();

        }

        protected void btnDelete\_Click(object sender, EventArgs e)

        {

            using (SampleDataContext dbContext = new SampleDataContext())

            {

                Employee employee = dbContext.Employees.SingleOrDefault(x => x.ID == 8);

                dbContext.Employees.DeleteOnSubmit(employee);

                dbContext.SubmitChanges();

            }

            GetData();

        }

    }

}

**Part 3 - How to view LINQ to SQL generated SQL queries**

**Suggested Videos**  
[Part 1 - LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-1-linq-to-sql.html)   
[Part 2 - Insert Update Delete using LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-2-insert-update-delete-using-linq.html)   
  
   
  
In this video we will discuss, **how to view the SQL statements generated by LINQ to SQL**. For debugging, it is necessary to view the generated SQL statements. There are several ways to do this.   
  
   
  
**Using the Log property of the DataContext object**

using (SampleDataContext dbContext = new SampleDataContext())

{

    // Write the generated sql query to the webform

    dbContext.Log = Response.Output;

    // Write the generated sql query to the Console window

    // dbContext.Log = Console.Out;

    var linqQuery = from employee in dbContext.Employees

                    select employee;

    GridView1.DataSource = linqQuery;

    GridView1.DataBind();

}

**Using ToString() method**

using (SampleDataContext dbContext = new SampleDataContext())

{

    var linqQuery = from employee in dbContext.Employees

                    select employee;

    string sqlQuery = linqQuery.ToString();

    Response.Write(sqlQuery);

    GridView1.DataSource = linqQuery;

    GridView1.DataBind();

}

**Using GetCommand() method of DataContext object**

using (SampleDataContext dbContext = new SampleDataContext())

{

    var linqQuery = from employee in dbContext.Employees

                    select employee;

    Response.Write(dbContext.GetCommand(linqQuery).CommandText);

    Response.Write("<br/>");

    Response.Write(dbContext.GetCommand(linqQuery).CommandType);

    GridView1.DataSource = linqQuery;

    GridView1.DataBind();

}

Finally, we can also use **SQL profiler.**

**Part 4 - Using stored procedures with LINQ to SQL**

**Suggested Videos**  
[Part 1 - LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-1-linq-to-sql.html)   
[Part 2 - Insert Update Delete using LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-2-insert-update-delete-using-linq.html)   
[Part 3 - How to view LINQ to SQL generated SQL queries](http://csharp-video-tutorials.blogspot.com/2014/09/part-3-how-to-view-linq-to-sql.html)   
  
   
  
In this video, we will discuss **retrieving data using stored procedures with LINQ to SQL**. This is continuation to [Part 3](http://csharp-video-tutorials.blogspot.com/2014/09/part-3-how-to-view-linq-to-sql.html). Please watch [Part 3](http://csharp-video-tutorials.blogspot.com/2014/09/part-3-how-to-view-linq-to-sql.html) before proceeding.   
  
   
  
**Here are the steps for using stored procedure with LINQ to SQL**  
**Step 1 :**Create the stored procedure

Create procedure GetEmployees

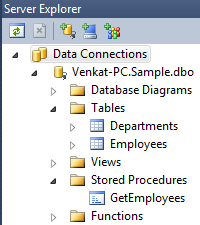
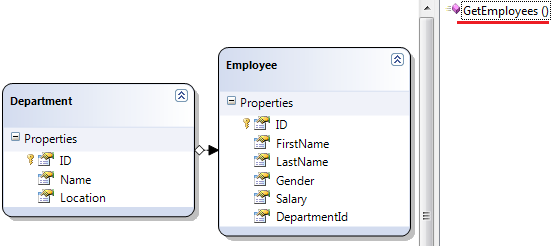
as

Begin

     Select ID, FirstName, LastName, Gender, Salary, DepartmentId

     from Employees

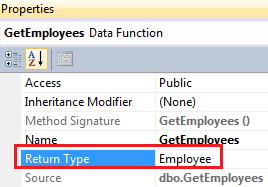
End

**Step 2 :** In Visual Studio, click on the **"View"**menu and select **"Server Explorer"**. Right click on **"Data Connection"**and  select **"Refresh".**Expand **"Stored Procedures"**folder. Here you should find **"GetEmployees"**stored procedure.   
   
  
**Step 3 :** Drag **"GetEmployees"**stored procedure from the **Server Explorer**window and drop it on the **LINQ to SQL class designer**. This will automatically create a method with the same name as the stored procedure. Tables and Views are converted to classes. Stored procedures and functions are converted to methods.   
   
  
**Step 4 :** Finally in the code-behind file, call **GetEmployees()**method on the **DataContext**class instance.

SampleDataContext dbContext = new SampleDataContext();

GridView1.DataSource = dbContext.GetEmployees();

GridView1.DataBind();

At this point, run SQL profiler and start a new trace. Run the application and click Get Data button. In the SQL profiler trace notice that the stored procedure is called as expected.  
  
**Modifying the Return Type of GetEmployees() method**  
Hover the mouse over **GetEmployees()**method and notice the Return Type. This method returns ISingleResult<GetEmployeesResult>   
return type of select stored procedure in linq to sql  
  
**There are 2 things that we need to understand here**  
**1.** GetEmployeesResult is an auto-generated type and follows the   
"[StoredProcedureName]Result" naming pattern.  
**2.** ISingleResult implies that we are getting back a single result set and not multiple result sets.   
  
**Can we change the Return Type**  
Yes, on the LINQ to SQL designer, right click on the GetEmployees() method and select properties. In the properties window set the return type to the type you are expecting. In this example, I have set it to Employee. So the return type now is **ISingleResult<Employee>**.  
  
Another way to do this is, when dragging and dropping the stored procedure on the designer surface, make sure to drop it on the Employee entity. Doing so will also set the return type to **ISingleResult<Employee>**.   


**Part 5 - Insert Update Delete using stored procedures in LINQ to SQL**

**Suggested Videos**  
[Part 2 - Insert Update Delete using LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-2-insert-update-delete-using-linq.html)  
[Part 3 - How to view LINQ to SQL generated SQL queries](http://csharp-video-tutorials.blogspot.com/2014/09/part-3-how-to-view-linq-to-sql.html)   
[Part 4 - Using stored procedures with LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-4-using-stored-procedures-with.html)   
  
   
  
In this video, we will discuss **using stored procedures to perform Insert Update and Delete.** This is continuation to [Part 4](http://csharp-video-tutorials.blogspot.com/2014/09/part-4-using-stored-procedures-with.html). Please watch [Part 4](http://csharp-video-tutorials.blogspot.com/2014/09/part-4-using-stored-procedures-with.html) before proceeding.   
  
   
  
**Step 1 :**Create **Insert, Update and Delete stored procedures**

-- Insert Stored Procedure

Create procedure InsertEmployee

@FirstName nvarchar(50),

@LastName nvarchar(50),

@Gender nvarchar(50),

@Salary int,

@DepartmentId int

as

Begin

     Insert into Employees(FirstName, LastName, Gender, Salary, DepartmentId)

     values (@FirstName, @LastName, @Gender, @Salary, @DepartmentId)

End

GO

-- Update Stored Procedure

Create procedure UpdateEmployee

@ID int,

@FirstName nvarchar(50),

@LastName nvarchar(50),

@Gender nvarchar(50),

@Salary int,

@DepartmentId int

as

Begin

     Update Employees Set

     FirstName = @FirstName, LastName = @LastName, Gender = @Gender,

     Salary = @Salary, DepartmentId = @DepartmentId

     where ID = @ID

End

GO

-- Delete Stored Procedure

Create procedure DeleteEmployee

@ID int

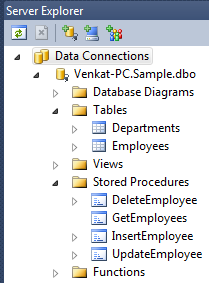
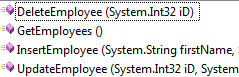
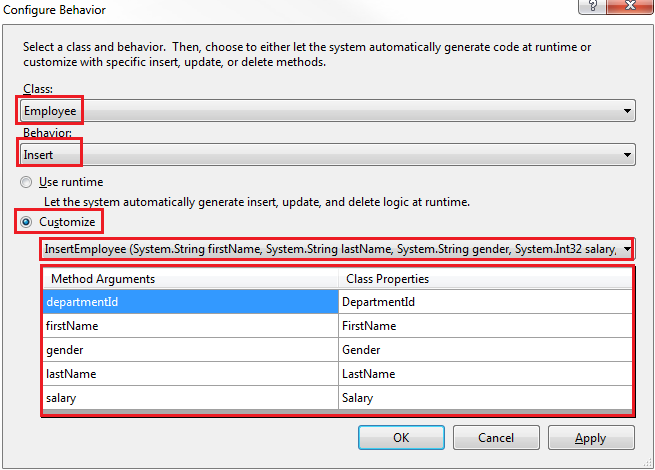
as

Begin

     Delete from Employees where ID = @ID

End

GO

**Step 2 :** In Visual Studio, click on the **"View"**menu and select **"Server Explorer"**. Right click on **"Data Connection"**and  select **"Refresh".**Expand **"Stored Procedures"**folder. Here you should find all the stored procedures.   
   
  
**Step 3 :** Drag **Insert, Update**and **Delete stored procedures**from the **Server Explorer**window and drop it on the **LINQ to SQL class designer.**This will automatically create respective methods with the same name as the stored procedures.  
   
  
**Step 4 :** Mapping stored procedures to work with LINQ to SQL  
**a)** Right click on **Employee**entity on LINQ to SQL designer and select **"Configure Behavior"**option.  
**b)** In the **"Configure Bevior"**window, set  
    Class = Employee  
    Behavior = Insert  
    Select "Customize" radio button  
    From the DropDownList, select InsertEmployee() stored procedure  
    Map Method Arguments to Class properties  
**c)** Finally click **OK**   
  
  
**d)** In a similar fashion, configure stored procedures for Update and Delete operations.  
  
At this point, run SQL profiler and start a new trace. Run the application and perform **Insert, Update and Delete**. In the SQL profiler trace notice that the respective stored procedures are called as expected.

**Part 6 - Stored procedures with output parameters in LINQ to SQL**

**Suggested Videos**  
[Part 3 - How to view LINQ to SQL generated SQL queries](http://csharp-video-tutorials.blogspot.com/2014/09/part-3-how-to-view-linq-to-sql.html)  
[Part 4 - Using stored procedures with LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-4-using-stored-procedures-with.html)   
[Part 5 - Insert Update Delete using stored procedures in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-5-insert-update-delete-using.html)   
  
   
  
In this video, we will discuss how to call stored procedures with output parameters using LINQ to SQL. This is continuation to [Part 5](http://csharp-video-tutorials.blogspot.com/2014/09/part-5-insert-update-delete-using.html). Please watch [Part 5](http://csharp-video-tutorials.blogspot.com/2014/09/part-5-insert-update-delete-using.html) before proceeding.   
  
   
  
**Step 1 : Create the stored procedure**

Create procedure GetEmployeesByDepartment

@DepartmentId int,

@DepartmentName nvarchar(50) out

as

Begin

     Select @DepartmentName = Name

     from Departments where ID = @DepartmentId

     Select \* from Employees

     where DepartmentId = @DepartmentId

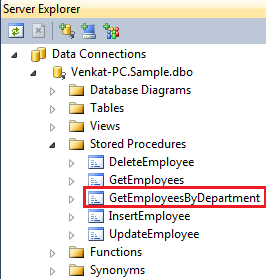
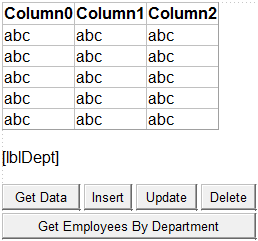
End

**Use the following SQL to test the stored procedure**

Declare @DeptName nvarchar(50)

Execute GetEmployeesByDepartment 2, @DeptName out

Select  @DeptName

**Step 2 :** In Visual Studio, click on the **"View"**menu and select **"Server Explorer".**Right click on **"Data Connection"**and  select **"Refresh".**Expand **"Stored Procedures"**folder. Here you should find **"GetEmployeesByDepartment"**stored procedure.   
   
  
**Step 3 :** Drag **"GetEmployeesByDepartment"**stored procedure from the **Server Explorer**window and drop it on the **LINQ to SQL class designer**. This will automatically create a method with the same name as the stored procedure.  
  
**Step 4 :**Drag and drop a button and a label control on the webform.   
  
**For the button**, change the following properties  
ID = lblDept  
Text=""  
  
**For the button**, change the following properties  
Text = Get Employees By Department  
ID = btnGetEmployeesByDepartment  
  
Double click the button control to generate the click event handler method.  
  
If you are following along the **design of the webform**, should be as shown below.  
   
  
**Step 5 :**Finally in the code-behind file, call **GetEmployeesByDepartment()**method using the **DataContext**class instance.

using (SampleDataContext dbContext = new SampleDataContext())

{

    string deptName = string.Empty;

    GridView1.DataSource = dbContext.GetEmployeesByDepartment(1, ref deptName);

    GridView1.DataBind();

    lblDept.Text = "Department Name = " + deptName;

}

**Part 7 - What is SqlMetal**

**Suggested Videos**  
[Part 4 - Using stored procedures with LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-4-using-stored-procedures-with.html)  
[Part 5 - Insert Update Delete using stored procedures in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-5-insert-update-delete-using.html)   
[Part 6 - Stored procedures with output parameters in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-6-stored-procedures-with-output.html)   
  
   
  
**In this video we will discuss**  
1. What is SqlMetal  
2. How to use SqlMetal   
  
   
  
**What is SqlMetal**  
**SqlMetal**is a command-line code generation tool used to generate **LINQ-to-SQL classes**. There are 2 ways to generate LINQ-to-SQL classes  
**1.** Uisng Visual Studio OR  
**2.** Using SqlMetal  
  
**What is the windows path where I can find SqlMetal.exe**  
On my machine SqlMetal.exe is present in the following location  
C:\Program Files (x86)\Microsoft SDKs\Windows\v7.0A\bin  
  
**How to use SqlMetal.exe to generate the LINQ-to-SQL classes**  
**Step 1 :**Run **Visual Studio Command prompt**as an **Administrator**  
  
**Step 2 :**In **C:\**create a new folder. Name it **SqlMetalGeneratedFiles.**This is the folder where we will store the **generated dbml**file.  
  
**Step 3 :**Type the following command and press enter  
  
**SqlMetal.exe  /server:localhost /database:Sample /namespace:Demo**  
**/dbml:C:\SqlMetalGeneratedFiles\Sample.dbml**  
**/Context:SampleDataContext**  
  
**In this example we are using the following options**  
**server -**Database server name. In our example, the database server is a local server, hence we specified localhost.  
**database -**database name  
  
**dbml -**The name of the generated dbml file  
  
**namespace -**Namespace for the generated classes  
  
**context -**Name of the data context class  
  
For the full list of all available options that can be used with SqlMetal.exe, please check the following MSDN article  
<http://msdn.microsoft.com/en-gb/library/vstudio/bb386987(v=vs.100).aspx>  
  
Navigate to **C:\SqlMetalGeneratedFiles\** and you should find **Sample.dbml**  
  
**Let's now discuss using Sample.dbml file in an ASP.NET Web Application.**  
**Step 1 :**Create a new empty asp.net web application project. Name it **Demo.**  
  
**Step 2 :**In the web.config file, copy and paste the following connection string

<add name="SampleConnectionString"

     connectionString="Data Source=venkat-pc;database=Sample;Integrated Security=True"

     providerName="System.Data.SqlClient" />

**Step 3 :**Right click on the **Demo**project, in solution explorer and select **Add - Existing Item.**Navigate to **C:\SqlMetalGeneratedFiles**and add **Sample.dbml.**  
  
**Step 4 :** Add a **WebForm**to the project. Drag and drop a **GridView**control on the **WebForm.**Copy and paste the following code in the code-behind file.

using System;

using System.Configuration;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        {

            string cs = ConfigurationManager.

                ConnectionStrings["SampleConnectionString"].ConnectionString;

            using (SampleDataContext dbContext = new SampleDataContext(cs))

            {

                GridView1.DataSource = dbContext.Employees;

                GridView1.DataBind();

            }

        }

    }

}

**Part 8 - Lazy loading in LINQ to SQL**

**Suggested Videos**  
[Part 5 - Insert Update Delete using stored procedures in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-5-insert-update-delete-using.html)  
[Part 6 - Stored procedures with output parameters in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-6-stored-procedures-with-output.html)   
[Part 7 - What is SqlMetal](http://csharp-video-tutorials.blogspot.com/2014/09/part-7-what-is-sqlmetal.html)   
  
   
  
In this video we will discuss the concept of **Lazy loading in LINQ to SQL with an example**.   
  
   
  
**Example 1 :**  
  
**Step 1 :**Create a database. Name it **Sample.**  
  
**Step 2 :**We will be using **Departments**and **Employees**tables in this demo. Create the tables using the following SQL script.

Create table Departments

(

     ID int primary key identity,

     Name nvarchar(50),

     Location nvarchar(50)

)

GO

Create table Employees

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50),

     Salary int,

     DepartmentId int foreign key references Departments(Id)

)

GO

Insert into Departments values ('IT', 'New York')

Insert into Departments values ('HR', 'London')

Insert into Departments values ('Payroll', 'Sydney')

GO

Insert into Employees values ('Mark', 'Hastings', 'Male', 60000, 1)

Insert into Employees values ('Steve', 'Pound', 'Male', 45000, 3)

Insert into Employees values ('Ben', 'Hoskins', 'Male', 70000, 1)

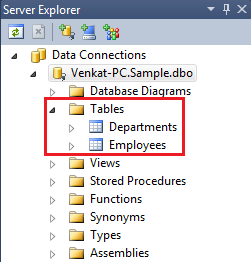
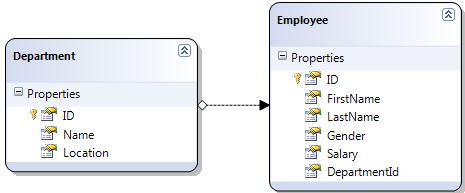
Insert into Employees values ('Philip', 'Hastings', 'Male', 45000, 2)

Insert into Employees values ('Mary', 'Lambeth', 'Female', 30000, 2)

Insert into Employees values ('Valarie', 'Vikings', 'Female', 35000, 3)

Insert into Employees values ('John', 'Stanmore', 'Male', 80000, 1)

GO

**Step 3 :** Create a new console application. Name it **ConsoleDemo.**  
  
**Step 4 :**Right click on the project in solution explorer and add **LINQ to SQL Classes**. Change the name from **DataClasses1.dbml**to **Sample.dbml**  
  
**Step 5 :**At this point, **Sample.dbml**file should have been added to the project. Click on **Server Explorer**link on **Sample.dbml**file. In the **Server Explorer**window, you should find all the tables in the **Sample**database. Drag and drop the tables on **Sample.dbml**file.   
   
  
**Step 6 :** At this point we should have **Department**and **Employee**classes. The properties of the class map to the columns of the respective table in the database. The arrow between the classes represent the association between them. These associations are modeled based on the primary-key/foreign-key relationships between the tables in the database. Notice that the arrow is pointing from Department to Employee entity. In this case there is a One-to-Many relationship between Department and Employee entities. A Department can have 1 or more employees.   
   
  
**Step 7 :** Copy and paste the following code in the **Main()**method in **Program.cs**file.

using (SampleDataContext dbContext = new SampleDataContext())

{

    //dbContext.Log = Console.Out;

    foreach (Department dept in dbContext.Departments)

    {

        Console.WriteLine(dept.Name);

        foreach (Employee emp in dept.Employees)

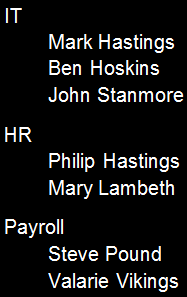
        {

            Console.WriteLine("\t" + emp.FirstName + " " + emp.LastName);

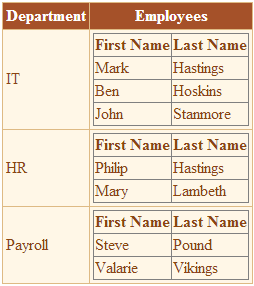
        }

    }

}

**Step 8 :** The above code prints, **Department Name**and all the employees **FirstName**and **LastName's**belonging to that **department.**The output is shown below.   
   
  
In this example there is a **One-to-Many relationship**between **Department**and **Employee**entities. A Department can have 1 or more employees. When Departments are loaded the related entities (Employee entities) are not loaded. Employee entities are only loaded when we iterate thru the employee objects belonging to a given department.  
  
To prove this, uncomment the following line and run the console application again. Notice that there is one query to retrieve all the departments and 3 queries to retrieve the employees belonging to each department.

//dbContext.Log = Console.Out;

**Example 2 :** Let us display **Department Name**and **Employee Names**in a GridView control as shown below in an asp.net web application.    
   
  
**Step 1 :** Create a new empty asp.net web application project.   
  
**Step 2 :**Name it **Demo.**Follow the steps in **Example 1**, from **Step 4** to **Step 6:**

**Step 3 :**Add a WebForm to the project. Copy and paste the following HTML in the aspx page.

<asp:GridView ID="gvDepartments" runat="server" AutoGenerateColumns="False">

    <Columns>

        <asp:BoundField HeaderText="Department" DataField="Name" />

        <asp:TemplateField HeaderText="Employees">

            <ItemTemplate>

                <asp:GridView ID="gvEmployees" runat="server"

                                AutoGenerateColumns="false"

                                DataSource='<%# Eval("Employees") %>'>

                    <Columns>

                        <asp:BoundField DataField="FirstName" HeaderText="First Name" />

                        <asp:BoundField DataField="LastName" HeaderText="Last Name" />

                    </Columns>

                </asp:GridView>

            </ItemTemplate>

        </asp:TemplateField>

    </Columns>

</asp:GridView>

**Step 4 :** Copy and paste the following code in the **Page\_Load()**event of the code-behind file.

using (SampleDataContext dbContext = new SampleDataContext())

{

    dbContext.Log = Response.Output;

    gvDepartments.DataSource = dbContext.Departments;

    gvDepartments.DataBind();

}

Run the application and notice that there is one query to retrieve all the departments and 3 queries to retrieve the employees belonging to each department. In this example we are data binding both departments and employees.  
  
**So, what is Lazy Loading**  
Lazy Loading means the related entities are not loaded until we iterate thru them or data bind them. By default, LINQ to SQL loads related entities by using Lazy Loading.  
  
In our next video, we will discuss how to change this behavior and eager load the related entities.

**Part 9 - Eager loading in LINQ to SQL**

**Suggested Videos**  
[Part 6 - Stored procedures with output parameters in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-6-stored-procedures-with-output.html)  
[Part 7 - What is SqlMetal](http://csharp-video-tutorials.blogspot.com/2014/09/part-7-what-is-sqlmetal.html)   
[Part 8 - Lazy loading in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-8-lazy-loading-in-linq-to-sql.html)   
  
   
  
In this video we will discuss **eager loading in LINQ to SQL**. This is continuation to [Part 8](http://csharp-video-tutorials.blogspot.com/2014/09/part-8-lazy-loading-in-linq-to-sql.html). Please watch [Part 8](http://csharp-video-tutorials.blogspot.com/2014/09/part-8-lazy-loading-in-linq-to-sql.html) before proceeding.   
  
   
  
**What is Eager loading**  
**Eager loading**is the process whereby a query for one type of entity also loads related entities as part of the query.   
  
**In LINQ to SQL there are 2 ways we can eager load data**  
**1.** Using DataLoadOptions   
**2.** Using Projection   
  
**Using DataLoadOptions to eager load related entities in LINQ to SQL :**

using (SampleDataContext dbContext = new SampleDataContext())

{

    dbContext.Log = Console.Out;

    // Load related Employee entities along with the Department entity

    DataLoadOptions loadOptions = new DataLoadOptions();

    loadOptions.LoadWith<Department>(d => d.Employees);

    dbContext.LoadOptions = loadOptions;

    foreach (Department dept in dbContext.Departments)

    {

        Console.WriteLine(dept.Name);

        foreach (Employee emp in dept.Employees)

        {

            Console.WriteLine("\t" + emp.FirstName + " " + emp.LastName);

        }

    }

}

**DataLoadOptions**is present in **System.Data.Linq** namespace  
  
Run the application, and notice that there is only one query which retrieves all the departments and their related employee entities.  
  
**Using Projection to eager load related entities in LINQ to SQL :**

using (SampleDataContext dbContext = new SampleDataContext())

{

    dbContext.Log = Console.Out;

    var linqQuery = from dept in dbContext.Departments

                    select new { Name = dept.Name, Employees = dept.Employees };

    foreach (var dept in linqQuery)

    {

        Console.WriteLine(dept.Name);

        foreach (Employee emp in dept.Employees)

        {

            Console.WriteLine("\t" + emp.FirstName + " " + emp.LastName);

        }

    }

}

Again, run the application, and notice that there is only one query which retrieves all the departments and their related employee entities.  
  
Now let's do the same thing with the web application example we worked with in [Part 8](http://csharp-video-tutorials.blogspot.com/2014/09/part-8-lazy-loading-in-linq-to-sql.html).  
  
**Using DataLoadOptions to eager load related entities in LINQ to SQL :**

using (SampleDataContext dbContext = new SampleDataContext())

{

    dbContext.Log = Response.Output;

    DataLoadOptions loadOptions = new DataLoadOptions();

    loadOptions.LoadWith<Department>(d => d.Employees);

    dbContext.LoadOptions = loadOptions;

    gvDepartments.DataSource = dbContext.Departments;

    gvDepartments.DataBind();

}

**Using Projection to eager load related entities in LINQ to SQL :**

using (SampleDataContext dbContext = new SampleDataContext())

{

    dbContext.Log = Response.Output;

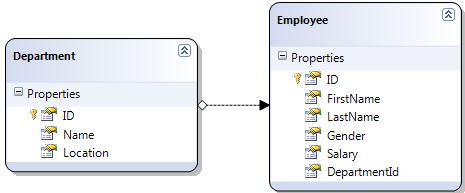
    gvDepartments.DataSource = from dept in dbContext.Departments

                                select new { Name = dept.Name, Employees = dept.Employees };

    gvDepartments.DataBind();

}

**Part 10 - Difference between eager loading and lazy loading**

**Suggested Videos**  
[Part 7 - What is SqlMetal](http://csharp-video-tutorials.blogspot.com/2014/09/part-7-what-is-sqlmetal.html)  
[Part 8 - Lazy loading in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-8-lazy-loading-in-linq-to-sql.html)   
[Part 9 - Eager loading in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-9-eager-loading-in-linq-to-sql.html)   
  
   
  
In this video we will discuss the **difference between eager loading and lazy loading**. This is continuation to [Part 9](http://csharp-video-tutorials.blogspot.com/2014/09/part-9-eager-loading-in-linq-to-sql.html). Please watch [Part 9](http://csharp-video-tutorials.blogspot.com/2014/09/part-9-eager-loading-in-linq-to-sql.html) before proceeding.   
  
   
  
With lazy loading there is a problem called **n + 1 select problem**. Let us understand this problem with an example. In this example there is a One-to-Many relationship between Department and Employee entities. A Department can have 1 or more employees.   
   
  
Now, let's say we need to iterate through all the Departments, and for each Department, we want to print the list of the employees. By default, LINQ to SQL would do the following:

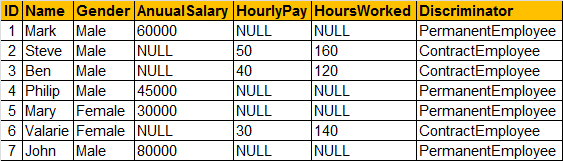
Select \* from Departments

/\* For each Department \*/

SELECT \* FROM Employees WHERE DepartmentId = X

So, this means we have one select for the Departments, and then N additional selects to retrieve the employees belonging to each Department, where N is the total number of Departments. So, this is N + 1 problem.  
  
**What is the difference between eager loading and lazy loading? Which is good - eager loading or lazy loading?**  
Without looking at the application architecture and what we are trying to achieve, we cannot say one is better over the other. Both have their own advantages and disadvantages. There are clear performance trade-offs between eager and lazy loading objects from a database.  
  
With **eager loading**, all the data is retrieved in a single query, which can then be cached to improve the application performance. With eager loading we are trading memory consumption for database round trips.  
  
With **lazy loading**, we only retrieve just the amount of data that we need in a single query. When we need more data related to the initial data, additional queries are issued to the database. This means there are several round trips between the application server and the database server. In general, these database round trips are very often the major performance bottleneck in most applications. Lesser the round trips, better the performance.  
  
**For example**, if on a given page you are only displaying Departments, then there is no reason for eager loading related Employees data. So in this case lazy loading works best. On the other hand, if you are displaying both Department and Employees data, then eager loading works best, as it avoids the additional round trips to the database.  
  
If you are not sure of what data is exactly needed, start with lazy loading and if it is leading to N + 1 problem then eager load the data.

**Part 11 - Single table inheritance in linq to sql**

**Suggested Videos**  
[Part 8 - Lazy loading in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-8-lazy-loading-in-linq-to-sql.html)  
[Part 9 - Eager loading in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-9-eager-loading-in-linq-to-sql.html)   
[Part 10 - Difference between eager loading and lazy loading](http://csharp-video-tutorials.blogspot.com/2014/09/part-10-difference-between-eager.html)   
  
   
  
With **single table inheritance** one database table is used to store data for all of the entity types in the entire inheritance hierarchy. Let us understand this with an example.   
  
   
  
We will be using the following **Employees**table.   
   
  
**SQL Script to create Employees Table**

Create Table Employees

(

     ID int primary key identity,

     Name nvarchar(50),

     Gender nvarchar(50),

     AnuualSalary int,

     HourlyPay int,

     HoursWorked int,

     Discriminator nvarchar(50)

)

GO

Insert into Employees values ('Mark', 'Male', 60000, NULL, NULL,'PermanentEmployee')

Insert into Employees values ('Steve', 'Male', NULL, 50, 160, 'ContractEmployee')

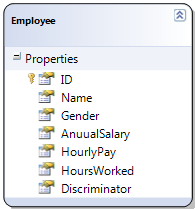
Insert into Employees values ('Ben', 'Male', NULL, 40, 120, 'ContractEmployee')

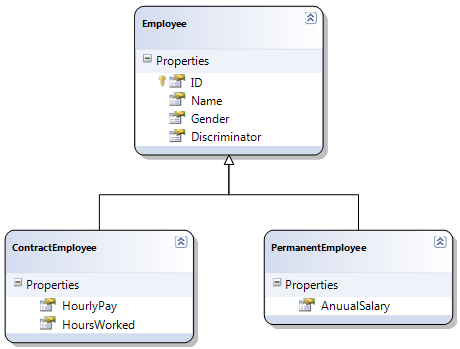
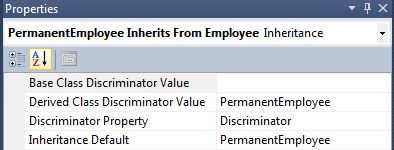
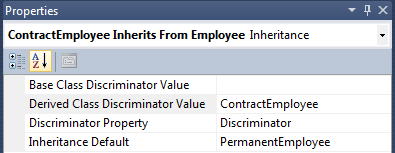
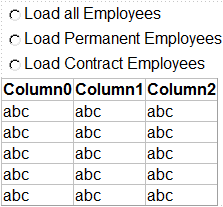
Insert into Employees values ('Philip', 'Male', 45000, NULL, NULL,'PermanentEmployee')

Insert into Employees values ('Mary', 'Female', 30000, NULL, NULL,'PermanentEmployee')

Insert into Employees values ('Valarie', 'Female', NULL, 30, 140, 'ContractEmployee')

Insert into Employees values ('John', 'Male', 80000, NULL, NULL, 'PermanentEmployee')

**In our organization we have 2 types of employees**  
**1. Permanent Employees -**AnuualSalary is specific for Permanent Employees  
**2. Contract Employees -**HourlyPay & HoursWorked is specific for Contract Employees  
  
By default LINQ to SQL creates the following Employee entity class, based on Employees table   
   
  
**But from an application perspective we want 3 entity classes**  
**Employee -**This should be an abstract class and should contain all the common properties of permanent and contract employees. ID, Name & Gender properties should be present in this class.

**PermanentEmployee -**This class should inherit from the abstract Employee class and should contain AnnualSalary property  
  
**ContractEmployee -**This class should inherit from the abstract Employee class and should contain HourlyPay & HoursWorked properties  
  
**To achieve this using the LINQ to SQL class designer**  
**1.** Right click on the designer surface and select **Add - Class**option. Change the class name to **PermanentEmployee**.  
  
**2.**Cut **AnnualSalary**property from **Employee**class and paste it in **PermanentEmployee**class.  
  
**3.** Right click on the designer surface and select **Add - Class**option. Change the class name to **ContractEmployee**.  
  
**4.**Cut **HourlyPay & HoursWorked**properties from **Employee**class and paste them in **ContractEmployee**class.  
  
**5.** Right click on **PermanentEmployee**and select **Add - Inheritance**option. In the **"New Inheritance"**window select **"Employee"**class from **"Select a base class"**dropdownlist and **"PermanentEmployee"**class from **"Select a derived class"**dropdownlist. This should create an inheritance relationship between **Employee**and **PermanentEmployee**classes.  
  
**6.** Along the same lines, Right click on **ContractEmployee**and select **Add - Inheritance**option. In the **"New Inheritance"**window select **"Employee"**class from **"Select a base class"**dropdownlist and **"ContractEmployee"**class from **"Select a derived class"**dropdownlist. This should create an inheritance relationship between **Employee**and **ContractEmployee**classes. At this point, the three classes should be as shown below.   
   
  
**7.** Right click on the **inheritance relationship arrow**(that connects Employee and PermanentEmployee classes) and select **"Properties".**Set the properties of the inheritance relationship as shown below.  
   
  
**8.**Along the same lines, right click on the inheritance relationship arrow(that connects Employee and ContractEmployee classes) and select "Properties". Set the properties of the inheritance relationship as shown below.   
   
  
**9.** Finally right click on **Employee**class and select **properties.**In the properties window set **Inheritance Modifier = abstract**. This should make Employee class an abstract class.   
  
Now, let us see how to query the data. Design a webform as shown below.    
   
  
**Here is the HTML for the web form.**

<divstyle="font-family: Arial">

<asp:RadioButtonListID="RadioButtonList1"runat="server"AutoPostBack="True"

onselectedindexchanged="RadioButtonList1\_SelectedIndexChanged">

<asp:ListItemText="Load all Employees"Value="All"></asp:ListItem>

<asp:ListItemText="Load Permanent Employees"Value="Permanent">

</asp:ListItem>

<asp:ListItemText="Load Contract Employees"Value="Contract">

</asp:ListItem>

</asp:RadioButtonList>

<asp:GridViewID="GridView1"runat="server">

</asp:GridView>

</div>

**Here is the code behind code:**

using System;

using System.Collections.Generic;

using System.Data;

using System.Linq;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        { }

        private DataTable ConvertEmployeesForDisplay(List<Employee> employees)

        {

            DataTable dt = new DataTable();

            dt.Columns.Add("ID");

            dt.Columns.Add("Name");

            dt.Columns.Add("Gender");

            dt.Columns.Add("AnuualSalary");

            dt.Columns.Add("HourlyPay");

            dt.Columns.Add("HoursWorked");

            dt.Columns.Add("Type");

            foreach (Employee employee in employees)

            {

                DataRow dr = dt.NewRow();

                dr["ID"] = employee.ID;

                dr["Name"] = employee.Name;

                dr["Gender"] = employee.Gender;

                if (employee is PermanentEmployee)

                {

                    dr["AnuualSalary"] = ((PermanentEmployee)employee).AnuualSalary;

                    dr["Type"] = "Permanent";

                }

                else

                {

                    dr["HourlyPay"] = ((ContractEmployee)employee).HourlyPay;

                    dr["HoursWorked"] = ((ContractEmployee)employee).HoursWorked;

                    dr["Type"] = "Contract";

                }

                dt.Rows.Add(dr);

            }

            return dt;

        }

        protected void RadioButtonList1\_SelectedIndexChanged(object sender, EventArgse)

        {

            SampleDataContext dbContext = new SampleDataContext();

            dbContext.Log = Response.Output;

            switch (RadioButtonList1.SelectedValue)

            {

                case "Permanent":

                    GridView1.DataSource =

                        dbContext.Employees.OfType<PermanentEmployee>().ToList();

                    GridView1.DataBind();

                    break;

                case "Contract":

                    GridView1.DataSource =

                        dbContext.Employees.OfType<ContractEmployee>().ToList();

                    GridView1.DataBind();

                    break;

                default:

                    GridView1.DataSource =

                        ConvertEmployeesForDisplay(dbContext.Employees.ToList());

                    GridView1.DataBind();

                    break;

            }

        }

    }

}

Run the application and inspect the queries that are generated.  
  
**When Load All Employees radio button is selected:**

SELECT [t0].[Discriminator], [t0].[HourlyPay], [t0].[HoursWorked], [t0].[ID], [t0].[Name], [t0].[Gender], [t0].[AnuualSalary] FROM [dbo].[Employees] AS [t0]

**When Load Permanent Employees radio button is selected:**

SELECT [t0].[Discriminator], [t0].[AnuualSalary], [t0].[ID], [t0].[Name], [t0].[Gender] FROM [dbo].[Employees] AS [t0] WHERE ([t0].[Discriminator] <> @p0) OR ([t0].[Discriminator] IS NULL) -- @p0: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [ContractEmployee]

**When Load Contract Employees radio button is selected:**

SELECT [t0].[Discriminator], [t0].[HourlyPay], [t0].[HoursWorked], [t0].[ID], [t0].[Name], [t0].[Gender] FROM [dbo].[Employees] AS [t0] WHERE ([t0].[Discriminator] = @p0) AND ([t0].[Discriminator] IS NOT NULL) -- @p0: Input NVarChar (Size = 4000; Prec = 0; Scale = 0) [ContractEmployee]

**art 12 - Single table inheritance - saving to database**

**Suggested Videos**  
[Part 9 - Eager loading in LINQ to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-9-eager-loading-in-linq-to-sql.html)  
[Part 10 - Difference between eager loading and lazy loading](http://csharp-video-tutorials.blogspot.com/2014/09/part-10-difference-between-eager.html)   
[Part 11 - Single table inheritance in linq to sql](http://csharp-video-tutorials.blogspot.com/2014/09/part-11-single-table-inheritance-in.html)   
  
   
  
In [Part 11](http://csharp-video-tutorials.blogspot.com/2014/09/part-11-single-table-inheritance-in.html), we discussed creating a single table inheritance model and retrieving data. In this video we will discuss saving data to the database table using the single table inheritance model. We will continue with the example we worked with in [Part 11](http://csharp-video-tutorials.blogspot.com/2014/09/part-11-single-table-inheritance-in.html).   
  
   
  
**Drag and drop a button control on the webform and set the following properties**  
ID="btnAddEmployees"  
Text="Add Employees"  
  
**Double click on the button control** to generate the click event handler. Copy and paste the following code in the code-behind file.

protected void btnAddEmployees\_Click(object sender, EventArgs e)

{

    using (SampleDataContext dbContext = new SampleDataContext())

    {

        PermanentEmployee permanentEmployee = new PermanentEmployee

        {

            Name = "Emma",

            Gender = "Female",

            AnuualSalary = 65000

        };

        ContractEmployee contractEmployee = new ContractEmployee

        {

            Name = "Kristie",

            Gender = "Female",

            HourlyPay = 50,

            HoursWorked = 80

        };

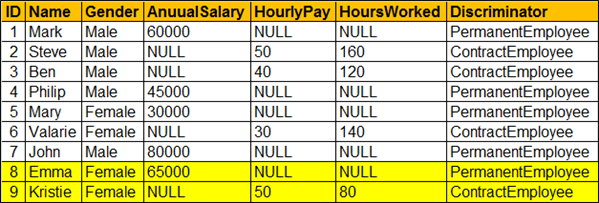
        dbContext.Employees.InsertOnSubmit(permanentEmployee);

        dbContext.Employees.InsertOnSubmit(contractEmployee);

        dbContext.SubmitChanges();

    }

}

Run the application and click **"Add Employees"**button  
  
**Please note:**  
**1.** Since **Employee**is an abstract class, an instance of this class cannot be created.  
**2.** The **Discriminator**column in the database is updated depending on the type of the employee object **(PermanentEmployee or ContractEmployee)**being instantiated.   
  


**Part 13 - Compiled queries in Linq to SQL**

**Suggested Videos**  
[Part 10 - Difference between eager loading and lazy loading](http://csharp-video-tutorials.blogspot.com/2014/09/part-10-difference-between-eager.html)  
[Part 11 - Single table inheritance in linq to sql](http://csharp-video-tutorials.blogspot.com/2014/09/part-11-single-table-inheritance-in.html)   
[Part 12 - Single table inheritance - saving to database](http://csharp-video-tutorials.blogspot.com/2014/09/part-12-single-table-inheritance-saving.html)   
  
   
  
In this video we will discuss **compiling Linq queries to improve performance**. First let us understand what happens when a LINQ query is issued with an example.   
  
   
  
The following Linq query retrieves a single student whose ID = 1.

using (SampleDataContext dbContext = new SampleDataContext())

{

    Student student = (from s in dbContext.Students

                                   where s.ID == 1

                                   select s).Single();

    Console.WriteLine(student.FirstName + " " + student.LastName);

}

When this LINQ query is issued, **LINQ has to parse the expression tree and dynamically generate the required T-SQL statements.** This happens every time a LINQ query is issued. This overhead can be removed by compiling the linq query.  
  
To **compile a LINQ query**use **CompiledQuery**class that is present in **System.Data.Linq**namespace.

var compiledStudentQuery = CompiledQuery.Compile(

                            (SampleDataContext dataContext, int studentId) =>

                                (from s in dataContext.Students

                                 where s.ID == 1

                                 select s).Single());

using (SampleDataContext dbContext = new SampleDataContext())

{

    Student student = compiledStudentQuery(dbContext, 1);

    Console.WriteLine(student.FirstName + " " + student.LastName);

}

**Here is the SQL to create Students table**

Create Table Students

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50)

)

GO

Insert into Students values ('Mark', 'Hastings', 'Male')

Insert into Students values ('Steve', 'Pound', 'Male')

Insert into Students values ('Ben', 'Hoskins', 'Male')

Insert into Students values ('Philip', 'Hastings', 'Male')

Insert into Students values ('Mary', 'Lambeth', 'Female')

GO

After the **Students**table is created, add a new **LINQ to SQL class**to the console project. Drag and drop **Students**table from **Server Explorer**onto **LINQ to SQL class designer**file.

**Part 14 - How to directly execute sql queries using Linq to SQL**

**Suggested Videos**  
[Part 11 - Single table inheritance in linq to sql](http://csharp-video-tutorials.blogspot.com/2014/09/part-11-single-table-inheritance-in.html)  
[Part 12 - Single table inheritance - saving to database](http://csharp-video-tutorials.blogspot.com/2014/09/part-12-single-table-inheritance-saving.html)   
[Part 13 - Compiled queries in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-13-compiled-queries-in-linq-to-sql.html)   
  
   
  
So far in this video series, we have not written any sql queries to retrieve data using linq to sql. We write a linq query, and the underlying LINQ to SQL provider dynamically generates the T-SQL required. If we want to have complete over the SQL that is executed against the database, then we can use **ExecuteQuery()**or **ExecuteCommand()**methods of the **DataContext**class.    
  
   
  
In this video we will discuss how to use **ExecuteQuery()**and **ExecuteCommand()**methods to directly execute SQL queries. Let us understand this with an example.  
  
**Step 1 :**Create Students table

Create Table Students

(

     ID int primary key identity,

     FirstName nvarchar(50),

     LastName nvarchar(50),

     Gender nvarchar(50)

)

GO

Insert into Students values ('Mark', 'Hastings', 'Male')

Insert into Students values ('Steve', 'Pound', 'Male')

Insert into Students values ('Ben', 'Hoskins', 'Male')

Insert into Students values ('Philip', 'Hastings', 'Male')

Insert into Students values ('Mary', 'Lambeth', 'Female')

GO

**Step 2 :** Create a new **Console application**. Name it Demo.  
  
**Step 3 :**Add a new LINQ to SQL class to the console project. Drag and drop **Students**table from Server Explorer onto LINQ to SQL class designer file.  
  
**Step 4 :**  
The following query retrieves all Male students from Students table

Select \* from Students where Gender='Male'

Use the following code to execute the above query using DataContext object's ExecuteQuery() method

using (SampleDataContext dbContext = new SampleDataContext())

{

    IEnumerable<Student> students = dbContext.ExecuteQuery<Student>(

        "Select \* from Students where Gender='Male'");

    foreach (Student student in students)

    {

        Console.WriteLine(student.FirstName + " " + student.LastName);

    }

}

In the above example, we have hard-coded Gender. If you want to parameterize the query, then use the following syntax.

using (SampleDataContext dbContext = new SampleDataContext())

{

    IEnumerable<Student> students = dbContext.ExecuteQuery<Student>(

        "Select \* from Students where Gender={0}", "Male");

    foreach (Student student in students)

    {

        Console.WriteLine(student.FirstName + " " + student.LastName);

    }

}

If you want to perform an **Insert, Update or Delete** then use **ExecuteCommand()**method. This method returns the number of rows affected by the query. The following code updates all 4 male student's gender to Female.

using (SampleDataContext dbContext = new SampleDataContext())

{

    int count = dbContext.ExecuteCommand(

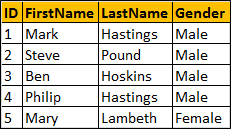
        "Update Students set Gender='Female' where Gender='Male'");

    Console.WriteLine("Rows Updated = {0}", count);

}

**Is it a good practice to use ExecuteQuery() or ExecuteCommand() methods to directly execute SQL queries?**  
No, use these methods only if absolutely necessary, that is when LINQ to SQL is not able to generate optimal SQL queries that you are expecting. In most of the cases LINQ to SQL does a pretty decent job in generating optimal sql queries. When we use ExecuteQuery() or ExecuteCommand() methods we loose the expressive power of LINQ and the advantage of having strongly-typed variables in queries.  
  
**What is the difference between ExecuteQuery and ExecuteCommand methods in linq**  
ExecuteQuery is used to perform a Select, while ExecuteCommand is used to perform Insert, Update, Delete or for calling a stored procedure.

**Part 15 - Identity Cache in Linq to SQL**

**Suggested Videos**  
[Part 12 - Single table inheritance - saving to database](http://csharp-video-tutorials.blogspot.com/2014/09/part-12-single-table-inheritance-saving.html)  
[Part 13 - Compiled queries in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-13-compiled-queries-in-linq-to-sql.html)   
[Part 14 - How to directly execute sql queries using Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-14-how-to-directly-execute-sql.html)   
  
   
  
In this video we will discuss **what is Identity cache and it's impact on Linq to SQL queries**.   
  
   
  
**What is Identity Cache**  
**DataContext**object uses Identity Cache to manage objects. Let us understand this with an example. We will be using the following students table in this demo.   
   
  
**References:**  
<http://msdn.microsoft.com/en-us/library/vstudio/dd627203(v=vs.100).aspx>  
  
**In this example**, we are retrieving two students from the database with the same identity and using the the same datacontext object. With the first LINQ query is executed, it gets translated to T-SQL. The SQL query is executed and the result is brought back into the application, where a Student object is created. The object identity is stored in the Identity cache. When the second LINQ query is issued for the same student object, LINQ checks the identity cache and returns a reference to the student object that already exists. Notice that there is only one call to the database, inspite of  having 2 linq queries. S1 and S2 are pointing to the same student object in memory.

using (SampleDataContext dbContext = new SampleDataContext())

{

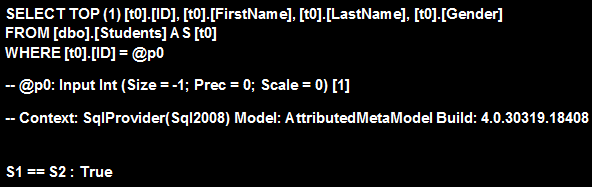
    dbContext.Log = Console.Out;

    Student S1 = dbContext.Students.FirstOrDefault(x => x.ID == 1);

    Student S2 = dbContext.Students.FirstOrDefault(x => x.ID == 1);

    Console.WriteLine("S1 == S2 : {0}", object.ReferenceEquals(S1, S2));

}

**Output:**   
  
  
**Each instance of LINQ to SQL DataContext class has its own identity cache**. This means if we have 2 different DataContext instances, and when we issue 2 linq queries 2 retrieve a student with the same identity, we get 2 different student objects back. S1 and S2 are two different objects in memory and registered in two different identity maps. Notice that the database also gets called 2 times.

using (SampleDataContext dbContext1 = new SampleDataContext())

using (SampleDataContext dbContext2 = new SampleDataContext())

{

    dbContext1.Log = Console.Out;

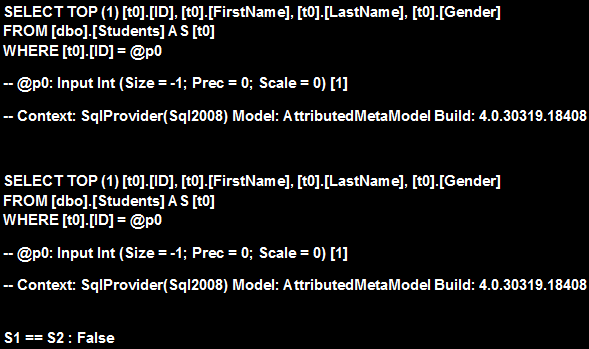
    dbContext2.Log = Console.Out;

    Student S1 = dbContext1.Students.FirstOrDefault(x => x.ID == 1);

    Student S2 = dbContext2.Students.FirstOrDefault(x => x.ID == 1);

    Console.WriteLine("S1 == S2 : {0}", object.ReferenceEquals(S1, S2));

}

**Output :**   
  
  
In this example we changed the FirstName of the student using the first datacontext object. We then called SubmitChanges() method, so the database has stored the new name for this student. However the student object for datacontext 2 still has the old name, because when we reissue the linq query for the same student object, this object is retrieved from the identity cache and not from the database. To have data retrieved from the database and to refresh the cache with updated values, we need to call the Refresh method of the second datacontext object.

using (SampleDataContext dbContext1 = new SampleDataContext())

using (SampleDataContext dbContext2 = new SampleDataContext())

{

    dbContext1.Log = Console.Out;

    dbContext2.Log = Console.Out;

    Student S1 = dbContext1.Students.FirstOrDefault(x => x.ID == 1);

    Student S2 = dbContext2.Students.FirstOrDefault(x => x.ID == 1);

    Console.WriteLine("S1.FirstName = {0}", S1.FirstName);

    Console.WriteLine("S2.FirstName = {0}", S2.FirstName);

    S1.FirstName = "Updated.....";

    dbContext1.SubmitChanges();

    Console.WriteLine("FirstName updated.....");

    S2 = dbContext2.Students.FirstOrDefault(x => x.ID == 1);

    Console.WriteLine("S1.FirstName = {0}", S1.FirstName);

    Console.WriteLine("S2.FirstName = {0}", S2.FirstName);

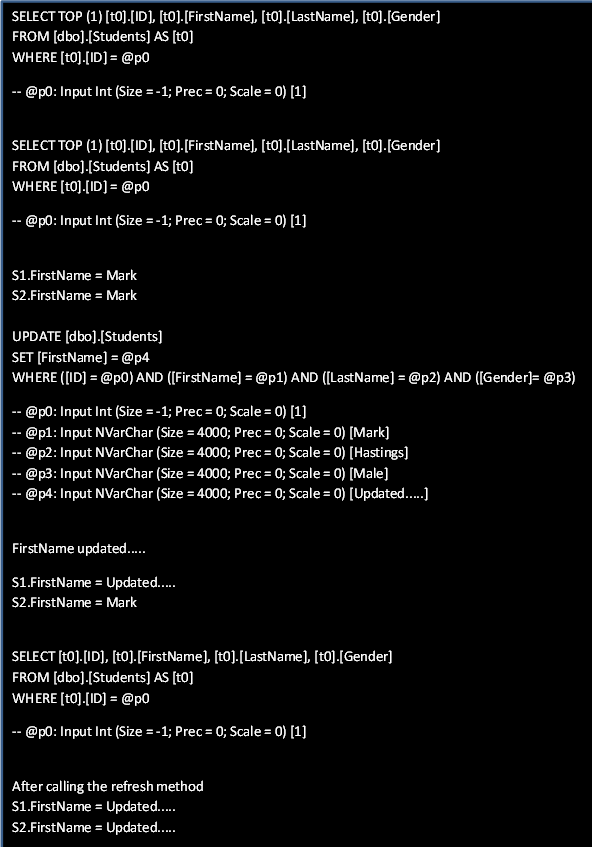
    dbContext2.Refresh(System.Data.Linq.RefreshMode.OverwriteCurrentValues, S2);

    Console.WriteLine("After calling the refresh method");

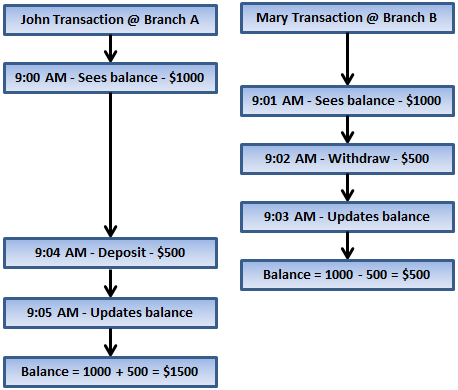
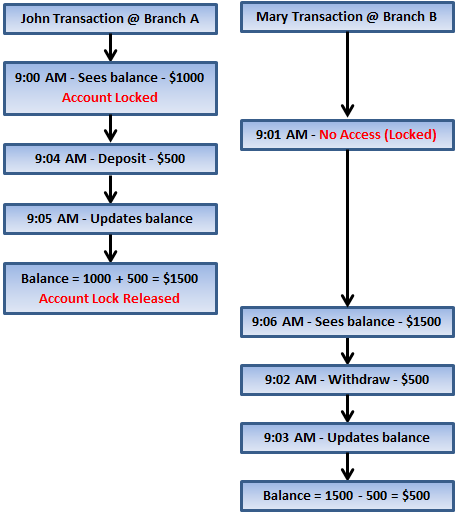
    Console.WriteLine("S1.FirstName = {0}", S1.FirstName);

    Console.WriteLine("S2.FirstName = {0}", S2.FirstName);

}

**Output :**   


**Part 16 - Difference between optimistic and pessimistic concurrency control**

**Suggested Videos**  
[Part 13 - Compiled queries in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-13-compiled-queries-in-linq-to-sql.html)  
[Part 14 - How to directly execute sql queries using Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-14-how-to-directly-execute-sql.html)   
[Part 15 - Identity Cache in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-15-identity-cache-in-linq-to-sql.html)   
  
   
  
**In this video we will discuss**  
**1.** Why is concurrency control required  
**2.** Difference between optimistic and pessimistic concurrency    
  
   
  
**Why is concurrency control required**  
Concurrency control is required to prevent two users from trying to update the same data at the same time. It can also prevent one user from seeing out-of-date data while another user is updating the same data.  
  
Let us understand **what can happen if there is no concurrency control in place** with an example.   
  
John and Mary has a joint account. At the moment the balance in the account is $1000. John and Mary visits different branches. John wants to deposit $500 and Mary wants to withdraw $500.  
  
The following are the transactions at the bank branches without concurrency control   
   
  
At the end of both the transaction the account balance must be $1000 (1000 - 500 + 500), but the balance now is $1500 which is incorrect. This happened because 2 users updated the balance at the same time and since there is no concurrency control in place, the second update has overwritten the changes made by the first update. This is a concurrency related problem and is called as **Lost updates problem**. There are several other concurrency related problems which we will discuss in a later video session.  
  
With the same example, let us now understand what can happen if there is some sort of concurrency control in place.  
  
The following are the transactions at the bank branches with concurrency control in place   
   
  
In this example, the account is locked while John is processing his transaction. The lock is released only after John's transaction is finished. After the lock has been released, Mary can proceed with her transaction. Since we have a concurrency control in place, we prevented 2 users from updating the balance at the same time which also prevented lost updates. So, the balance is updated correctly as expected.  
  
**There are 2 different concurrency control mechanisms**  
**1.** Pessimistic concurrency control   
**2.** Optimistic concurrency control  
  
**What is the difference between optimistic and pessimistic concurrency control**  
**Pessimistic concurrency**involves locking rows to prevent other users from modifying the same data at the same time. Until the lock is released by the lock owner, no other users will be able to access that data. Pessimistic locking can very easily lead to performance bottle necks in an application.  
  
**Optimistic concurrency** does not involve locking rows when reading. Instead, this model checks if two users tried to update the same record at the same time. If that happens one user's changes are committed and the other user's changes are discarded and an exception will be thrown to notify the user.  
  
We will discuss **how Linq to SQL implements optimistic concurrency** with an example in our next video.

**Part 17 - Concurrency in Linq to SQL**

**Suggested Videos**  
[Part 14 - How to directly execute sql queries using Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-14-how-to-directly-execute-sql.html)  
[Part 15 - Identity Cache in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-15-identity-cache-in-linq-to-sql.html)   
[Part 16 - Difference between optimistic and pessimistic concurrency control](http://csharp-video-tutorials.blogspot.com/2014/10/part-17-concurrency-in-linq-to-sql.html)   
  
   
  
This is continuation to [Part 16](http://csharp-video-tutorials.blogspot.com/2014/10/part-16-difference-between-optimistic.html). Please watch [Part 16](http://csharp-video-tutorials.blogspot.com/2014/10/part-16-difference-between-optimistic.html) before proceeding.  
  
In this video we will discuss **how linq to sql handles concurrent updates i.e when 2 or more users try to update the same data at the same time**.    
  
   
  
By default, **linq to sql uses optimistic concurrency to handle concurrent updates**. Optimistic concurrency does not involve locking rows when reading. Instead, this model checks if two users tried to update the same record at the same time. If that happens one user's changes are committed and the other user's changes are discarded and an exception will be thrown to notify the user.  
  
Let us look at this in action with an example. We will be using the following **Accounts**table in this demo.   
concurrency control in linq to sql   
  
**Step 1 :** Create the Accounts table

Create Table Accounts

(

     AccountNumber int primary key,

     AccountName nvarchar(50),

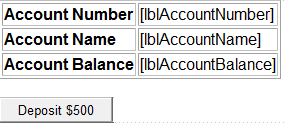
     AccountBalance int

)

Go

Insert into Accounts values (1, 'John Mary', 1000)

Go

**Step 2 :** Create a new empty asp.net web application project. Name it Demo. Add a new Linq to SQL Class. Name it Sample. Drag and drop Accounts table on Sample.dbml file.  
  
**Step 3 :**Add a WebForm and design it as shown below.   
   
  
**Here is the HTML for the web form**.

<div style="font-family:Arial">

<table border="1">

    <tr>

        <td>

            <b>Account Number</b>

        </td>

        <td>

            <asp:Label ID="lblAccountNumber" runat="server"></asp:Label>

        </td>

    </tr>

    <tr>

        <td>

            <b>Account Name</b>

        </td>

        <td>

            <asp:Label ID="lblAccountName" runat="server"></asp:Label>

        </td>

    </tr>

    <tr>

        <td>

            <b>Account Balance</b>

        </td>

        <td>

            <asp:Label ID="lblAccountBalance" runat="server"></asp:Label>

        </td>

    </tr>

</table>

<br />

<asp:Button ID="btnDeposit" runat="server" Text="Deposit $500"

    onclick="btnDeposit\_Click" />

</div>

**Step 4 :** Copy and paste the following code in **WebForm1.aspx.cs**

using System;

using System.Linq;

namespace Demo

{

    public partial class WebForm1 : System.Web.UI.Page

    {

        protected void Page\_Load(object sender, EventArgs e)

        {

            if (!IsPostBack)

            {

                GetAccountData();

            }

        }

        private void GetAccountData()

        {

            using (SampleDataContext db = new SampleDataContext())

            {

                Account account = db.Accounts.First(x => x.AccountNumber == 1);

                lblAccountNumber.Text = account.AccountNumber.ToString();

                lblAccountName.Text = account.AccountName;

                lblAccountBalance.Text = account.AccountBalance.ToString();

            }

        }

        protected void btnDeposit\_Click(object sender, EventArgs e)

        {

            using (SampleDataContext db = new SampleDataContext())

            {

                Account account = db.Accounts.First(x => x.AccountNumber == 1);

                account.AccountBalance = account.AccountBalance + 500;

                db.SubmitChanges();

                GetAccountData();

            }

        }

    }

}

At this point open **SQL Profiler** and run a **new trace**. Run the web application. Click **"Deposit $500"**button. Notice that the balance is updated to 1500 as expected.   
  
**Now inspect the Update query that is generated by linq to sql**

exec sp\_executesql N'UPDATE [dbo].[Accounts]

SET [AccountBalance] = @p3

WHERE ([AccountNumber] = @p0) AND ([AccountName] = @p1)

AND ([AccountBalance] = @p2)',

N'@p0 int,@p1 nvarchar(4000),@p2 int,@p3 int',@p0=1,

@p1=N'John Mary',@p2=1000,@p3=1500

To update the balance of the account all we need is the **New Balance**and the **AccountNumber**that needs to be updated. In the **WHERE**clause of the query notice that along with **AccountNumber**and the **New Balance**, we also have **AccountName**and the **original AccountBalance**. The reason we have these is to make sure that no column values in the row have changed since we have loaded the data from the database. If any of the values have changed, then the update fails and an exception will be thrown.  
  
**Let's now simulate the scenario of 2 users updating the same record at the same time**. To do this

**Step 1 :**Throw a break point on the line where we call **db.SubmitChanges();** in **btnDeposit\_Click()** method.  
  
**Step 2 :**Run the application in **Debug mode**, and click **"Deposit $500"** button. The processing should stop just before we call SubmitChanges() method.  
  
**Step 3 :**At this point open SQL Server Management Studio, and execute the following query

Update Accounts set AccountBalance = AccountBalance - 500

Where AccountNumber = 1

**Step 4 :** Now come back to Visual Studio and **press F5**to continue the execution. Notice that **"ChangeConflictException"** is thrown and the exception message states - Row not found or changed.

**Part 18 - Handling ChangeConflictException**

**Suggested Videos**  
[Part 15 - Identity Cache in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/09/part-15-identity-cache-in-linq-to-sql.html)  
[Part 16 - Difference between optimistic and pessimistic concurrency control](http://csharp-video-tutorials.blogspot.com/2014/10/part-16-difference-between-optimistic.html)  
[Part 17 - Concurrency in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/10/part-17-concurrency-in-linq-to-sql.html)   
  
   
  
This is continuation to [Part 17](http://csharp-video-tutorials.blogspot.com/2014/10/part-17-concurrency-in-linq-to-sql.html). Please watch [Part 17](http://csharp-video-tutorials.blogspot.com/2014/10/part-17-concurrency-in-linq-to-sql.html) before proceeding. In this video we will discuss how to **handle ChangeConflictException.**    
  
   
  
There are 3 options available to **handle ChangeConflictException**. RefreshMode enum values define how to handle optimistic concurrency conflicts. This enum has 3 values  
  
**KeepCurrentValues -** Keeps all the current changes made by the current user in the DataContext object. **SubmitChanges()**method will save all changes made by the current user, overwriting any changes made by other users after the data was loaded by the current user.  
  
**KeepChanges -**Keeps the current values that have been changed, but updates the other values with the database values. SubmitChanges() method will save any changes made by the current user and will preserve any changes made by other users. If another user changed the same value as the current user, the current user's change will overwrite it.  
  
**OverwriteCurrentValues -**Updates the DataContext with the current database values, which means that all changes made by the current user will be discarded.  
  
**Example :**To handle the exception include the Linq to Sql code in try/catch block. Modify the code in **btnDeposit\_Click()**as shown below.

protected void btnDeposit\_Click(object sender, EventArgs e)

{

    using (SampleDataContext db = new SampleDataContext())

    {

        try

        {

            Account account = db.Accounts.First(x => x.AccountNumber == 1);

            account.AccountBalance = account.AccountBalance + 500;

            db.SubmitChanges();

            GetAccountData();

        }

        catch (ChangeConflictException)

        {

            db.ChangeConflicts.ResolveAll(RefreshMode.OverwriteCurrentValues);

            foreach (ObjectChangeConflict objectChangeConflict

                in db.ChangeConflicts)

            {

                foreach (MemberChangeConflict memberChangeConflict

                    in objectChangeConflict.MemberConflicts)

                {

                    Response.Write("Current Value = " +

                        memberChangeConflict.CurrentValue.ToString() + "<br/>");

                    Response.Write("Original Value = " +

                        memberChangeConflict.OriginalValue.ToString() + "<br/>");

                    Response.Write("Database Value = " +

                        memberChangeConflict.DatabaseValue.ToString() + "<br/>");

                }

            }

            db.SubmitChanges();

            GetAccountData();

        }

    }

}

**With the above changes**  
**Step 1 :** Throw a break point on the line where we call db.SubmitChanges(); in btnDeposit\_Click() method.  
  
**Step 2.**Run the application in Debug mode, and click "Deposit $500" button. The processing should stop just before we call SubmitChanges() method.  
  
**Step 3 :**At this point open SQL Server Management Studio, and execute the following query

Update Accounts set AccountBalance = AccountBalance - 300

Where AccountNumber = 1

**Step 4 :**Now come back to Visual Studio and press F5 to continue the execution. Notice that **"ChangeConflictException"**is thrown and handled. Also, the balance gets overwritten with the current value.  
  
Now change the following line  
db.ChangeConflicts.ResolveAll(RefreshMode.KeepCurrentValues);  
TO  
db.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);  
  
Run the application again in Debug mode, and click "Deposit $500" button. When the application excution stops at the break point. Execute the following UPDATE statement from SQL Server Management Studio. The SQL statement has updated 2 columns (AccountName and AccountBalance)

Update Accounts set AccountBalance = AccountBalance - 300,

AccountName = 'John-Mary' Where AccountNumber = 1

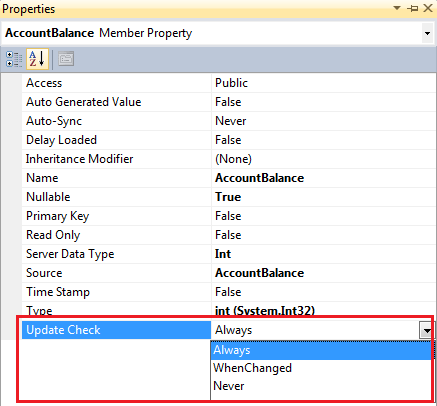
Now come back to Visual Studio and press F5 to continue the execution. Notice that "ChangeConflictException" is thrown and handled. The balance gets overwritten with the current value, but the AccountName is not. This is because RefreshMode.KeepChanges option will keep and update only the values that have been changed by the current user since the data is loaded into the DataContext. This means SubmitChanges() will save only changes made by the current user and will preserve any changes made by other users.   
  
Now change the following line  
db.ChangeConflicts.ResolveAll(RefreshMode.KeepChanges);  
TO  
db.ChangeConflicts.ResolveAll(RefreshMode.OverwriteCurrentValues);  
  
Run the application again in Debug mode, and click "Deposit $500" button. When the application excution stops at the break point. Execute the following UPDATE statement from SQL Server Management Studio. The SQL statement has changed AccountName and AccountBalance.

Update Accounts set AccountBalance = AccountBalance - 300,

AccountName = 'Mary John' Where AccountNumber = 1

Now come back to Visual Studio and press F5 to continue the execution. Notice that "ChangeConflictException" is thrown and handled. Notice that the changes made by the other user are preserved. So, RefreshMode.OverwriteCurrentValues will update the DataContext with the current database values, which means that all changes made by the current user will be discarded.

**Part 19 - UpdateCheck property**

**Suggested Videos**  
[Part 16 - Difference between optimistic and pessimistic concurrency control](http://csharp-video-tutorials.blogspot.com/2014/10/part-16-difference-between-optimistic.html)  
[Part 17 - Concurrency in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/10/part-17-concurrency-in-linq-to-sql.html)   
[Part 18 - Handling ChangeConflictException](http://csharp-video-tutorials.blogspot.com/2014/10/part-18-handling-changeconflictexception.html)   
  
   
  
This is continuation to [Part 18](http://csharp-video-tutorials.blogspot.com/2014/10/part-18-handling-changeconflictexception.html). Please watch [Part 18](http://csharp-video-tutorials.blogspot.com/2014/10/part-18-handling-changeconflictexception.html), before proceeding. **UpdateCheck**property of **ColumnAttribute**class is used to determine how LINQ to SQL detects concurrency conflicts.   
  
   
  
**To set this property**  
**1.** Open **Sample.dbml**file  
**2.** Right click on **AccountBalance**property and select **"Properties"**  
**3.** In the properties window set a value for the **UpdateCheck**property   
   
  
This property can be set to one of the 3 values of the **UpdateCheck**enum. This enum is present in **System.Data.Linq.Mapping** namespace. The following are the different values of UpdateCheck enum and what they mean.

|  |  |
| --- | --- |
| **Always** | Always use this column for conflict detection |
| **Never** | Never use this column for conflict detection |
| **WhenChanged** | Use this column only when the member has been changed by the application |

The default is Always. This is means by default all the columns will be used to detect concurrency conflicts.  
  
**Example :**  
**1.** Open Sample.dbml file and right click on "AccountBalance" property and select "Properties" option from the context menu.   
  
**2.** In the properties window set, UpdateCheck = Never. At this point open, Sample.Designer.cs file and notice that the AccountBalance property has UpdateCheck property applied.

[global::System.Data.Linq.Mapping.ColumnAttribute

    (Storage="\_AccountBalance", DbType="Int", UpdateCheck=UpdateCheck.Never)]

public System.Nullable<int> AccountBalance

{

     get

     {

          return this.\_AccountBalance;

     }

     set

     {

          if ((this.\_AccountBalance != value))

          {

              this.OnAccountBalanceChanging(value);

              this.SendPropertyChanging();

              this.\_AccountBalance = value;

              this.SendPropertyChanged("AccountBalance");

              this.OnAccountBalanceChanged();

          }

     }

}

3. Open SQL Profiler and run a new trace  
  
4. Run the application and click "Deposit $500" button  
  
5. Inspect the generated UPDATE SQL command.

exec sp\_executesql N'UPDATE [dbo].[Accounts]

SET [AccountBalance] = @p2

WHERE ([AccountNumber] = @p0) AND ([AccountName] = @p1)',

N'@p0 int,@p1 nvarchar(4000),@p2 int',@p0=1,@p1=N'John Mary',@p2=2200

Notice, that **AccountBalance**is removed from the WHERE clause, which means this column is not used for detecting concurrency conflicts.

**Part 20 - Using ROWVERSION or TIMESTAMP to detect concurrency conflicts**

**Suggested Videos**  
[Part 17 - Concurrency in Linq to SQL](http://csharp-video-tutorials.blogspot.com/2014/10/part-17-concurrency-in-linq-to-sql.html)  
[Part 18 - Handling ChangeConflictException](http://csharp-video-tutorials.blogspot.com/2014/10/part-18-handling-changeconflictexception.html)   
[Part 19 - UpdateCheck Property](http://csharp-video-tutorials.blogspot.com/2014/10/part-19-updatecheck-property.html)   
  
   
  
This is continuation to [Part 19](http://csharp-video-tutorials.blogspot.com/2014/10/part-19-updatecheck-property.html). Please watch [Part 19](http://csharp-video-tutorials.blogspot.com/2014/10/part-19-updatecheck-property.html) before proceeding. In this video we will discuss how to use **ROWVERSION**or **TIMESTAMP**columns to detect concurrency conflicts in linq to sql. Let us understand this with an example. We will be using the following **Accounts**table in this demo.   
  
   
  
http://2.bp.blogspot.com/-9pvULtESlUQ/VDWYTxZnIfI/AAAAAAAAWVM/fGibTWCmjOs/s1600/concurrency%2Bcontrol%2Bin%2Blinq%2Bto%2Bsql.png   
  
By default LINQ to SQL uses all the columns of the table in the WHERE clause to detect concurrency conflicts. The query would look as shown below.

exec sp\_executesql N'UPDATE [dbo].[Accounts]

SET [AccountBalance] = @p3

WHERE ([AccountNumber] = @p0) AND ([AccountName] = @p1)

AND ([AccountBalance] = @p2)',

N'@p0 int,@p1 nvarchar(4000),@p2 int,@p3 int',@p0=1,

@p1=N'John Mary',@p2=1000,@p3=1500

This is OK if we have a few columns in the table. In real time applications we may have tables with large number of columns. For example, what if the table has 30 columns. The WHERE clause would be huge and it can impact the performance of the application.  
  
In situations like this we can use **ROWVERSION**or **TIMESTAMP**columns. Here are the steps  
  
**Step 1 :**Add a **Version**column to the **Accounts**table. The datatype of the column must be either **ROWVERSION**or **TIMESTAMP.**The value for this column is automatically generated by the database if the row gets changed. So this column can alone be used to detect concurrency conflicts.

ALTER TABLE Accounts

ADD [Version] ROWVERSION

**Step 2 :** In Visual Studio, delete the **Account**Entity from the **Sample.dbml**file  
  
**Step 3 :**In Server Explorer window in Visual Studio, right click on **Accounts**table and select **"Refresh"**.   
  
**Step 4 :**Drag and  drop **Accounts**table on the Designer surface of **Sample.dbml**file. Notice that a **Version**Property is automatically added. Navigate to **Sample.Designer.cs**file and look at the code generated for this property. Notice that **IsVersion** & **IsDbGenerated**properties are set to true.

[global::System.Data.Linq.Mapping.ColumnAttribute(Storage = "\_Version",

AutoSync = AutoSync.Always, DbType = "rowversion NOT NULL", CanBeNull = false,

IsDbGenerated = true, IsVersion = true, UpdateCheck = UpdateCheck.Never)]

public System.Data.Linq.Binary Version

{

    get

    {

        return this.\_Version;

    }

    set

    {

        if ((this.\_Version != value))

        {

            this.OnVersionChanging(value);

            this.SendPropertyChanging();

            this.\_Version = value;

            this.SendPropertyChanged("Version");

            this.OnVersionChanged();

        }

    }

}

**Testing for concurrency conflicts :** Let's now simulate the scenario of 2 users updating the same record at the same time. To do this  
  
**Step 1 :**Throw a break point on the line where we call db.SubmitChanges(); in btnDeposit\_Click() method.  
  
**Step 2 :**Run the application in Debug mode, and click "Deposit $500" button. The execution should stop on SubmitChanges() method.  
  
**Step 3 :**At this point open SQL Server Management Studio.   
   a) Execute the following SELECT query  
       Select \* from Accounts where AccountNumber = 1  
   b) Notice the Value of Version column  
   c) Execute the following UPDATE query  
       Update Accounts set AccountBalance = AccountBalance - 300   
       Where AccountNumber = 1  
   d) Now notice that the Value of Version column is automatically changed to a new value  
  
**Step 4 :** Open SQL Profiler and run a new trace.  
  
**Step 5 :**Now come back to Visual Studio and press F5 to continue the execution. Notice that **"ChangeConflictException"**is thrown as expected.  
  
**Step 6 :**In SQL Profiler notice that the UPDATE query that is generated has used Version column in the WHERE clause to detect concurrency conflicts.

exec sp\_executesql N'UPDATE [dbo].[Accounts]

SET [AccountBalance] = @p2

WHERE ([AccountNumber] = @p0) AND ([Version] = @p1)

SELECT [t1].[Version]

FROM [dbo].[Accounts] AS [t1]

WHERE ((@@ROWCOUNT) > 0) AND ([t1].[AccountNumber] = @p3)',

N'@p0 int,@p1 timestamp,@p2 int,@p3 int',

@p0=1,@p1=0x0000000000002715,@p2=1500,@p3=1